ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Course	:	Full Stack Development
Course Code	:	**24FMCA31T**
Branch	:	**MCA**
Prepared by	:	P. Kavitha
Designation	:	**Assistant Professor**
Department	:	**MCA**

ANNAMACHARYA UNIVERSITY, RAJAMPET

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016
RAJAMPET, Annamayya District, AP, INDIA

Title of the Course : **Full Stack Development**
Category : **PC**
Year : **II**
Semester : **I**
Course Code : **24FMCA31T**
Branch : **MCA**

Lecture Hours	Tutorial Hours	Practice Hours	Credits
3	0	0	3

COURSE OBJECTIVES:

- Develop strong foundation in object oriented programming principles.
- Provide strong foundation in web development including HTML,CSS and XML Technologies.
- Understand basic concepts of Java Script.
- Understand Node JS Concepts and introduction of Express JS.
- Build on foundational knowledge of Mongo Data base connections.

COURSE OUTCOMES:
The Student will be able to

1. Apply exception handling mechanism using Java for developing applications.
2. Apply HTML, CSS and XML features for developing web pages.
3. Apply Java Script features for developing web pages.
4. Apply NODE JS for developing various web applications.
5. Apply Mongo DB for storing data in database.

UNIT I **12**

CLASSES AND OBJECTS: Concepts of classes, objects, methods, access control, this keyword, simple java program, constructors, parameter passing, recursion, Enumeration, String Handling, Inheritance, Polymorphism, Packages and Interfaces, Exception Handling.

UNIT II **12**

HTML COMMON TAGS: Introduction, HTML Basics- Text, Colors, Links, Images, Lists, Forms, Frames, Tables, Web Page Design, HTML5, Cascading Style Sheets, Introduction to CSS, Types of CSS.
XML: Introduction, Document Type Definition, XML Schema, Document Object Model, Presenting XML(XSL), XML Parsers: DOM and SAX.

UNIT III **12**

JAVA SCRIPT: Introduction to JavaScript, Declaring Java Script Variables, Basics of JavaScript, Control Structures, Pop Up Boxes, Functions, Arrays, Events, Objects.

UNIT IV **10**

NODE JS: Introduction to Node JS, Installation of Nods JS, Node JS File System Modules, NODE JS npm, Errors, Crypto Module, NODE JS Callbacks, Events, Web Modules. Introduction to Express JS, Express JS Installation, Express JS- GET, POST, REQUEST, RESPONSE, Express JS- Cookies, Middleware Routing.

UNIT V **8**

Introduction to Mongo DB, Differences between SQL and NOSQL, Mongo DB datatypes, Mongo DB installation, Data Modeling in Mongo DB- Create Database, Insert Mongo DB, Update Mongo DB,find Mongo DB , Delete Mongo Db, Queries in Mongo DB.

PRESCRIBED TEXTBOOKS:

1. Herbert Schildt, Java. The Complete Reference, 12th Edition, 2022
2. Uttam K Roy, Web Technologies , Oxford University Press, 1st Edition, 2010.
3. Nabendu Biswas, Full Stack Web Development with MERN, Orange Education Pvt Ltd, 1st Edition 2023.

REFERENCE BOOKS:

1. Y. Daniel Liang. Introduction to Java Programming, Pearson Education, 12th Edition, 2024.
2. Colin J Ihrig, Full Stack JavaScript Development with MEAN , Sitepoint Publishers,1st Edition, 2015.
3. Philip Ackermann, Full Stack Development(The Comprehensive Guide), Shroff/ Rheinwerk Computing, Gray Scale Edition, 2023.

CO-PO MAPPING:

Course Outcomes	Foundation Knowledge	Problem Analysis	Development of Solutions	Modern Tool Usage	Individual and Teamwork	Project Management and Finance	Ethics	Life-long Learning
24FMCA31T.1	2	2	1	-	-	-	-	-
24FMCA31T.2	3	2	1	-	-	-	-	-
24FMCA31T.3	3	2	1	-	-	-	-	-
24FMCA31T.4	3	2	1	-	-	-	-	-
24FMCA31T.5	3	2	1	-	-	-	-	-

UNIT I

CLASSES AND OBJECTS: Concepts of classes, objects, methods, access control, this keyword, simple java program, constructors, parameter passing, recursion, Enumeration, String Handling, Inheritance, Polymorphism, Packages and Interfaces, Exception Handling.

Java - Overview

•Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

•The latest release of the Java Standard Edition is Java SE 8. With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms.

•Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

•The new J2 versions were renamed as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere.**

Java is:

Object Oriented: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

• **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.

• **Secure:** With Java's secure feature it enables to develop virus free, tamper-free systems. Authentication techniques are based on public-key encryption.

• **Architectural-neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

• **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

• **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
• **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
• **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
• **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
• **Distributed:** Java is designed for the distributed environment of the internet.
• **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

• **History of Java:**
• James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later being renamed as Java, from a list of random words.
• Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere**(WORA), providing no-cost run-times on popular platforms.
• On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
• On 8 May 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.
• **Tools you will need:**
• For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).
• **You also will need the following softwares:**
• Linux 7.1 or Windows xp/7/8 operating system.
• Java JDK 8

• Microsoft Notepad or any other text editor

• **Java - Environment Setup**

Local Environment Setup

• If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

• Java SE is freely available from the link Download Java. So you download a version based on your operating system.

• Follow the instructions to download java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

• **Setting up the path for windows:**

• Assuming you have installed Java in *c:\Program Files\java\jdk*directory:

• Right-click on 'My Computer' and select 'Properties'.

• Click on the 'Environment variables' button under the 'Advanced' tab.

• Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

• **Setting up the path for Linux, UNIX, Solaris, FreeBSD**:

• Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

• Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

• **Popular Java Editors:**

• To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following:

• **Notepad:** On Windows machine you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.

• **Netbeans:** is a Java IDE that is open-source and free which can be downloaded from http://www.netbeans.org/index.html.

• **Eclipse:** is also a Java IDE developed by the eclipse open-source community and can be downloaded fromhttp://www.eclipse.org/.

• **Java - Basic Syntax**

• When we consider a Java program it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods and instance variables mean.

• **Object -** Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviours -wagging, barking, eating. An object is an instance of a class.

• **Class -** A class can be defined as a template/ blue print that describes the behaviours/states that object of its type support.

• **Methods -** A method is basically a behaviour. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

• **Instance Variables -** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

• **First Java Program:**

• Let us look at a simple code that would print the words *Hello World*.

```
public class MyFirstJavaProgram
{
/* This is my first java program.
* This will print 'Hello World' as the output
*/
public static void main(String[]args)
{
System.out.println("Hello World");
// prints Hello World
}
}
```

• Let's look at how to save the file, compile and run the program.

Please follow the steps given below:

• Open notepad and add the code as above.

• Save the file as: MyFirstJavaProgram.java.
• Open a command prompt window and go to the directory where
you saved the class. Assume it's C:\.• Type ' javac MyFirstJavaProgram.java' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption : The path variable is set).
• Now, type ' java MyFirstJavaProgram ' to run your program.
• You will be able to see ' Hello World ' printed on the window.

• C:\> javac MyFirstJavaProgram.java
• C:\> java MyFirstJavaProgram
• Hello World

• **Basic Syntax:**
• About Java programs, it is very important to keep in mind the following points.
• **Case Sensitivity -** Java is case sensitive, which means
identifier **Hello** and **hello** would have different meaning in Java.
• **Class Names -** For all class names the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.

Example *class MyFirstJavaClass*
• **Method Names -** All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
Example *public void myMethodName()*
• **Program File Name -** Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile).
Example: Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as*'MyFirstJavaProgram.java'*
• **public static void main(String args[]) -** Java program processing starts from the main( ) method which is a mandatory part of every Java program.

1.1 Classes

Java is an Object-Oriented Language. As a language that has the Object Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this , we will look into the concepts Classes and Objects.

1.2 Object - Objects have states and behaviors. Example: A dog has states -
color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

Class - A class can be defined as a template/blue print that describes the
behaviors/states that object of its type support.

• **Objects in Java:**

• Let us now look deep into what are objects. If we consider the realworld we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behavior.

• If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging, running

• If you compare the software object with a real world object, they
have very similar characteristics.

• Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods.

• So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

1.3 Access control is a mechanism, an attribute of encapsulation which restricts the **access** of certain members of a class to specific parts of a program. **Access** to members of a class can be **controlled** using the **access** modifiers. There are four **access** modifiers in **Java**.

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. **The four access levels are** −

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

1) Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example :Variables and methods can be declared without any modifiers, as in the following examples −

String version = "1.5.1"; boolean processOrder() { return true; }**2) Private Access 2) Modifier - Private**

- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
- Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.
- **Example**
- The following class uses private access control −

public class Logger

{

```
private String format;
public String getFormat()
{
return this.format;
}
public void setFormat(String format)
{
this.format = format; } }
```

• Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

• So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

3) **Public Access Modifier - Public**

• A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

• However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

• **Example**

• The following function uses public access control −

```
public static void main(String[] arguments)
{
// ...
}
```

• The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

4) Protected Access Modifier - Protected

• Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

• The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

• Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

• **Example**

• The following parent class uses protected access control, to allow its child class override *openSpeaker()* method −

```
class AudioPlayer
{
protected boolean openSpeaker(Speaker sp)
{
// implementation details
}
}
class StreamingAudioPlayer extends AudioPlayer
{
boolean openSpeaker(Speaker sp)
{
// implementation details
}
}
```

• Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than *AudioPlayer*.

• If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that’s why we have used protected modifier.

• **Access Control and Inheritance**

• The following rules for inherited methods are enforced −

• Methods declared public in a super class also must be public in all subclasses.

• Methods declared protected in a super class must either be protected or public in subclasses; they cannot be private.

• Methods declared private are not inherited at all, so there is no rule for them.

1.4 this keyword in java

• **Usage of java this keyword**

• Here is given the 6 usage of java this keyword.

1) this can be used to refer current class instance variable.

2) this can be used to invoke current class method (implicitly)

3) this() can be used to invoke current class constructor.

4) this can be passed as an argument in the method call.

5) this can be passed as argument in the constructor call.

6) this can be used to return the current class instance from the method.

• **1) this: to refer current class instance variable**

• The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

• **Understanding the problem without this keyword**

• Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
```

```
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display(); s2.display();
}
}
```

Output:

0 null 0.0 0

null 0.0•

In the above example, parameters (formal arguments) and instance variables are same.

So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
```

```
s1.display();
s2.display();
}
}
```

•

Output:

• 111 ankit 5000

• 112 sumit 6000

• If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

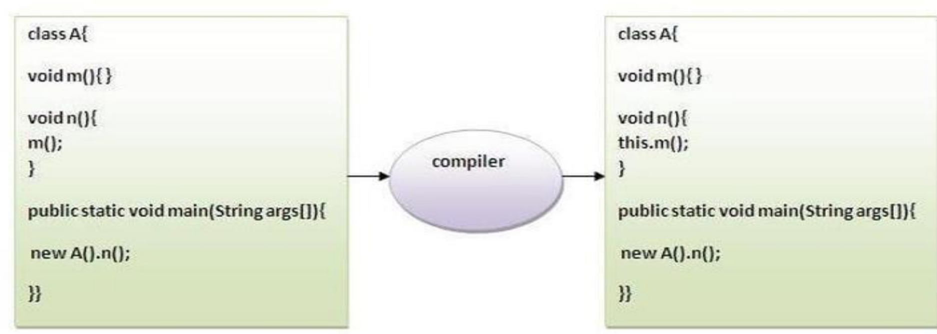
• **Program where this keyword is not required**

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis3{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f); Student s2=new Student(112,"sumit",6000f);
s1.display(); s2.display();
}
}
```

• **Output:**

• 111 ankit 5000

• 112 sumit 6000




```
class A{
void m()
{
System.out.println("hello m");
}
void n()
{
System.out.println("hello n");
//m(); //same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[])
{
A a=new A();
a.n();
}
}
```

• **Output:**

• hello n

• hello m

3) this() : to invoke current class constructor

• The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

• **Calling default constructor from parameterized constructor:**

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}
}
```

• **Output:**

• hello a 10

• **Calling parameterized constructor from default constructor:**

```
class A{
A( ){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
```

```
class TestThis6{
public static void main(String args[]){
A a=new A( );
}
}
```

• **Output:**

5 hello a

4) this: to pass as an argument in the method

• The this keyword can also be passed as an argument in the method.

It is mainly used in the event handling. Let's see the example:

```
class S2{
void m(S2 obj){
System.out.println("method is invoked");
}
void p(){
m(this);
}
public static void main(String args[]){
S2 s1 = new S2();
s1.p();
}
}
```

• **Output:**

method is invoked

5) this: to pass as argument in the constructor call

• We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
A4 obj;
B(A4 obj){
this.obj=obj;
```

```
}
void display(){
System.out.println(obj.data);//using data member of A4 class
}
}
class A4{
int data=10;
A4(){
B b=new B(this);
b.display();
}
public static void main(String args[]){
A4 a=new A4();
} }
```

Output:10

6) this keyword can be used to return current class instance

- We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:
- Syntax of this that can be returned as a statement
- return_type method_name(){
- **return this**;
- }

- Example of this keyword that you return as a statement from the method

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
```

```
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

Output:

• Hello java

1.5 First Java Program | Hello World Example

• We can write a simple hello java program easily after installing the JDK.

• To create a simple java program, you need to create a class that contains the main method.

• Let's understand the requirement first.

The requirement for Java Hello World Example

1) For executing any java program, you need toInstall the JDK if you don't have installed it, download the JDK and install it.

2) Set path of the jdk/bin directory.

http://www.javatpoint.com/how-to-set-path-in-java

3) Create the java program

4) Compile and run the java program• **Creating Hello World Example**

Let's create the hello java program:

```
class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}
```

• Save this file as Simple.java

• **To compile:** javac Simple.java

• **To execute:** java Simple

• **Output:** Hello Java

• **Parameters used in First Java Program**

• Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

• **class** keyword is used to declare a class in java.

• **public** keyword is an access modifier which represents visibility. It means it is visible to all.

• **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.

• **void** is the return type of the method. It means it doesn't return any value.

• **main** represents the starting point of the program.

• **String[] args** is used for command line argument. We will learn it later.

• **System.out.println()** is used to print statement. Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class. We will learn about the internal working of System.out.println statement later.

• To compile and run the above program, go to your current directory first; my current directory is • c:\new. Write here:

• **To compile:** javac Simple.java

• **To execute:** java Simple

• **How many ways can we write a Java program**

• There are many ways to write a Java program. The modifications that can be done in a Java program are given below:

1) By changing the sequence of the modifiers, method prototype is not changed in Java.

• Let's see the simple code of the main method.

static public void main(String args[])

2) The subscript notation in Java array can be used after type, before the variable or after the variable.

• Let's see the different codes to write the main method.

public static void main(String[ ] args)

public static void main(String [ ]args)

public static void main(String args[ ])

3) You can provide var-args support to the main method by passing 3 ellipses (dots)

• Let's see the simple code of using var-args in the main method. We will learn about var-args later in Java New Features chapter.

public static void main(String... args)

4) Having a semicolon at the end of class is optional in Java.

• Let's see the simple code.

```
class A{
static public void main(String... args){
System.out.println("hello java4");
}
};
```

Valid java main method signature

- **public static void** main(String[] args)
- **public static void** main(String []args)
- **public static void** main(String args[])
- **public static void** main(String... args)
- **static public void** main(String[] args)
- **public static final void** main(String[] args)
- **final public static void** main(String[] args)
- **final strictfp public static void** main(String[] args)

Invalid java main method signature

- **public void** main(String[] args)
- **static void** main(String[] args)
- **public void static** main(String[] args)
- **abstract public static void** main(String[] args)

• **Resolving an error "javac is not recognized as an internal or external command"?**

• If there occurs a problem like displayed in the below figure, you need to set path.

• Since DOS doesn't know javac or java, we need to set path.

• The path is not required in such a case if you save your program inside the JDK/bin directory.

• However, it is an excellent approach to set the path.

• Click here for How to set path in java.Right Click on Computer, Select Properties Option and then press enter Key.You will get the below window. Select Advanced System Settings,and Press enter Key.You will get this dialog box, Select Environment Variables.You will get this dialog box. In User Variables select New button. After selecting New Button, you will get this dialog box, then type the variable name as path , and copy variable value from C drive.Place the data like this and click ok button, then ok button and again ok button. Then java path has been set.

• **1.6** A **constructor in Java** is a special method that is used to initialize objects.

• The **constructor** is called when an object of a class is created.

• It can be used to set initial values for object attributes:

• A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

• Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures
required to create a fully formed object.

• All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero.

• However, once you define your own constructor, the default constructor is no longer used.

• **Syntax**

• Following is the syntax of a constructor −

• class ClassName {

•

ClassName()

• { }

• }

• **Java allows two types of constructors namely −**

1) No argument Constructors

2) Parameterized Constructors

1) **No argument Constructors**

• As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

• **Example**

```
public class MyClass {
int num;
MyClass() {
num = 100;
}
}
```

• **You would call constructor to initialize objects as follows**

```
public class ConsDemo {
public static void main(String args[]) {
MyClass t1 = new MyClass();
MyClass t2 = new MyClass();
System.out.println(t1.num + " " + t2.num);
}
}
```

• **This would produce the following result**

• 100 100

2) **Parameterized Constructors**

• Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

• **Example**

• Here is a simple example that uses a constructor −

• // A simple constructor.

```
class MyClass {
int x; // Following is the constructor
MyClass(int i ) {
x = i;
}
}
```

• **You would call constructor to initialize objects as follows −**

```
public class ConsDemo {
public static void main(String args[]) {
MyClass t1 = new MyClass( 10 );
MyClass t2 = new MyClass( 20 );
System.out.println(t1.x + " " + t2.x);
}
}
```

• **This would produce the following result −**

• 10 20

1.7 Parameter Passing Techniques in Java with Examples

• There are different ways in which parameter data can be passed into and out of methods and functions.

• Let us assume that a function *B()* is called from another function *A()*. In this case *A* is called the ***"caller function"*** and *B* is called the ***"called function or callee function"***.

• Also, the arguments which *A* sends to *B* are called *actual arguments* and the parameters of *B* are called *formal arguments*.

Types of parameters:

1) Formal Parameter : A variable and its type as they appear in the prototype of the function or method.

Syntax:function_name(datatype variable_name)

2) Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

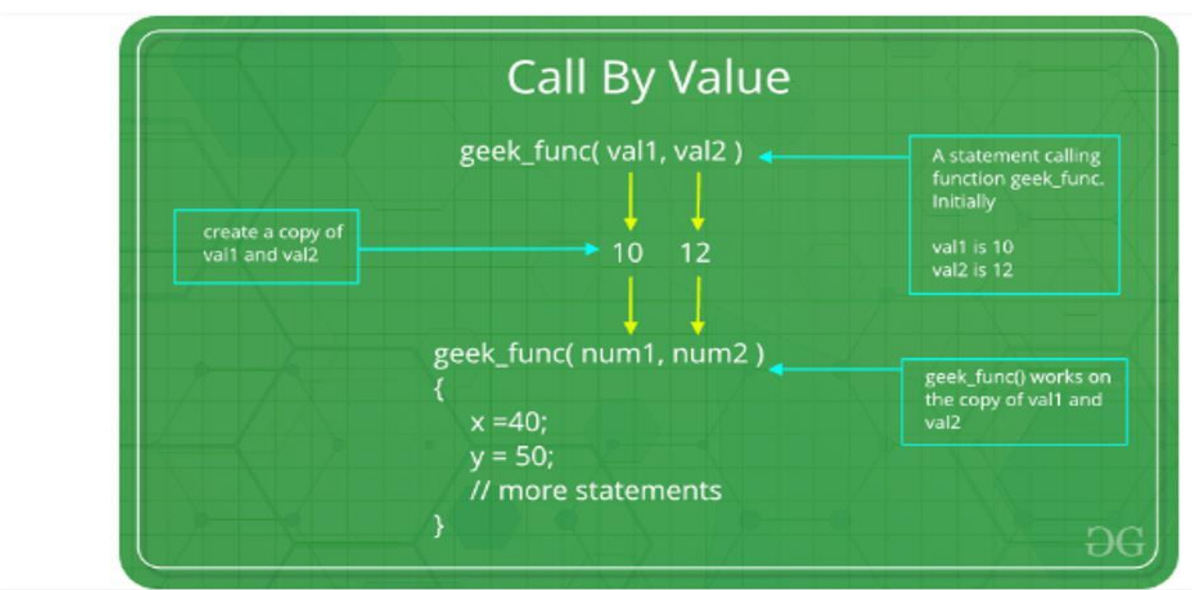
Syntax:func_name(variable name(s));

• **Important methods of Parameter Passing**

• **Pass By Value:** Changes made to formal parameter do not get transmitted back to the caller.

• Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment.

• This method is also called as ***call by value***.



• // Java program to illustrate

• // Call by Value

• // Callee

```
class CallByValue {
// Function to change the value of the parameters
public static void Example(int x, int y)
{
x++;
y++;
```

```
}
} •
// Caller
public class Main {
public static void main(String[] args)
{
int a = 10;
int b = 20;
// Instance of class is created
CallByValue object = new CallByValue();
System.out.println("Value of a: " + a
+ " & b: " + b);
// Passing variables in the class function
object.Example(a, b);
// Displaying values after
// calling the function
System.out.println("Value of a: "
+ a + " & b: " + b);
} }
```

Output:

Value of a: 10 & b: 20

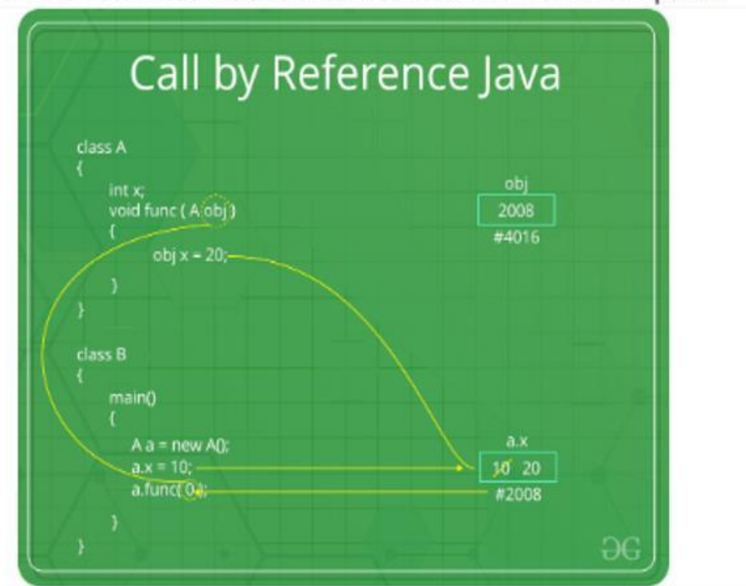
Value of a: 10 & b: 20

• **Shortcomings:**

• Inefficiency in storage allocation

• For objects and arrays, the copy semantics are costly•

2. **Call by reference(aliasing):** Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as **<em>call by reference**. This method is efficient in both time and space.



// Java program to illustrate

• // Call by Reference

• // Callee

class CallByReference {

int a, b;

// Function to assign the value to the class variables

CallByReference(int x, int y)

{

a = x;

b = y;

}

// Changing the values of class variables

void ChangeValue(CallByReference obj)

{

```
obj.a += 10;
obj.b += 20;
}
} •
// Caller
public class Main {
public static void main(String[] args)
{
// Instance of class is created and value is assigned using constructor
CallByReference object = new CallByReference(10, 20);
System.out.println("Value of a: "
+ object.a
+ " & b: "
+ object.b);
// Changing values in class function
object.ChangeValue(object);
// Displaying values
// after calling the function
System.out.println("Value of a: "
+ object.a
+ " & b: "
+ object.b);
}
}
```

Output: Value of a: 10 & b: 20

Value of a: 20 & b: 40• Please note that when we pass a reference, a new reference variable to the same object is created.

• So we can only change members of the object whose reference is passed.

• We cannot change the reference to refer to some other object as the received reference is a copy of the original reference.

1.8 Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in **java** that calls itself is called **recursive** method. It makes the code compact but complex to understand

.

How Recursion works?

```
public static void main(String[] args) {
    ... .. ...
    recurse()
    ... .. ...
}

static void recurse() {
    ... .. ...
    recurse()
    ... .. ...
}
```

Normal Method Call

Recursive Call

Working of Java Recursion

• In the above example, we have called the recurse() method from

inside the main method. (normal method call).

• And, inside the recurse() method, we are again calling the same recurse method. This is a recursive call.

• In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.

• Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

• **Example: Factorial of a Number Using Recursion**

```
class Factorial {
static int factorial( int n ) {
if (n != 0) // termination condition return n * factorial(n-1);
// recursive call
else
```

```
return 1;
}
public static void main(String[] args)
{
int number = 4, result;
result = factorial(number);
System.out.println(number + " factorial = " + result);
} }
```

• **Output**:

• 4 factorial = 24

• In the above example, we have a method named factorial().

The factorial() is called from the main() method. with the number variable passed as an argument.

• **Here, notice the statement,**

• return n * factorial(n-1);

• The factorial() method is calling itself. Initially, the value of n is 4 inside factorial(). During the next recursive call, 3 is passed to the factorial() method. This process continues until n is equal to 0.

• When n is equal to 0, the if statement returns false hence 1 is returned. Finally, the accumulated result is passed to the main() method.

• **Java Recursion Example 3: Factorial Number**

```
public class RecursionExample3 {
static int factorial(int n){
if (n == 1)
return 1;
else
return(n * factorial(n-1));
}
public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
```

}

• Output:

• Factorial of 5 is: 120

Working of above program:

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5*24 = 120
```

Java Recursion Example: Fibonacci Series

```
public class RecursionExample4 {
   static int n1=0,n2=1,n3=0;
    static void printFibo(int count){
      if(count>0){
          n3 = n1 + n2;
          n1 = n2;
          n2 = n3;
          System.out.print(" "+n3);
          printFibo(count-1);
       }
    }

public static void main(String[] args) {
   int count=15;
     System.out.print(n1+" "+n2);//printing 0 and 1
     printFibo(count-2);//n-2 because 2 numbers are already printed
}
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

1.9 • **Enumeration** means a list of named constant.

• In **Java**, **enumeration** defines a class type. An **Enumeration** can have constructors, methods and instance variables.

• It is created using **enum** keyword.

• The **Enum in Java** is a data type which contains a fixed set of constants.

• It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.

• According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

• Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).• The Java enum constants are static and final implicitly. It is available since JDK 1.5.
• Enums are used to create our own data type like classes.
The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*.
Here, we can define an enum either inside the class or outside the class.
• Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum

• Enum improves type safety
• Enum can be easily used in switch
• Enum can be traversed
• Enum can have fields, constructors and methods
• Enum may implement many interfaces but cannot extend any class because it internally extends Enum class.

• **Simple Example of Java Enum**

```
class EnumExample1{
//defining the enum inside the class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//main method
public static void main(String[] args) {
//traversing the enum
for (Season s : Season.values())
System.out.println(s);
}
}
```

• **Output:**

• WINTER SPRING SUMMER FALL•

Let us see another example of Java enum where we are using value(), valueOf(), and ordinal() methods of Java enum.

```
class EnumExample1{
//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL } //creating the main method
public static void main(String[] args) { //printing all enum
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());
} }
```

•

Output:

- WINTER
- SPRING
- SUMMER
- FALL
- Value of WINTER is: WINTER
- Index of WINTER is: 0
- Index of SUMMER is: 2
- **Note**: Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.
- **What is the purpose of the values() method in the enum?**
- The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.
- **What is the purpose of the valueOf() method in the enum?**
- The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

• **What is the purpose of the ordinal() method in the enum?**

• The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

• **Defining Java Enum**

• The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional. For example:

• **enum** Season { WINTER, SPRING, SUMMER, FALL }

•

Or,

• **enum** Season { WINTER, SPRING, SUMMER, FALL; }

• Both the definitions of Java enum are the same.

• **Java Enum Example: Defined outside class**

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}
}
```

• **Output:**

• WINTER

• **Java Enum Example: main method inside Enum**

• If you put main() method inside the enum, you can run the enum directly.

```
enum Season {
WINTER, SPRING, SUMMER, FALL;
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}
}
```

• **Output:**
• WINTER

• **Initializing specific values to the enum constants**
• The enum constants have an initial value which starts from 0, 1, 2, 3, and so on. But, we can initialize the specific value to the enum constants by defining fields and constructors.
As specified earlier, Enum can have fields, constructors, and methods.
• Example of specifying initial value to the enum constants

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);
private int value;
private Season(int value){
this.value=value;
}
}
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);
}
}
```

• **Output:**
• WINTER 5
• SPRING 10
• SUMMER 15
• FALL 20
• Constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

```
enum Season{
WINTER(10),SUMMER(20);
```

```
private int value;
Season(int value){
this.value=value;
}
}
```

• Internal code generated by the compiler for the above example of enum type

```
final class Season extends Enum
{
public static Season[] values()
{
return (Season[])$VALUES.clone(); }
public static Season valueOf(String s)
{
return (Season)Enum.valueOf(Season,s);
}
private Season(String s, int i, int j)
{
super(s, i);
value = j;
}
public static final Season WINTER;
public static final Season SUMMER;
private int value;
private static final Season $VALUES[];
static
{
WINTER = new Season("WINTER", 0, 10);
SUMMER = new Season("SUMMER", 1, 20);
$VALUES = (new Season[] {
WINTER, SUMMER
});
```

}
}

• **Can we create the instance of Enum by new keyword?**

• No, because it contains private constructors only.Can we have an abstract method in the Enum?

• Yes, Of course! we can have abstract methods and can provide the implementation of these methods.

• **Java Enum in a switch statement**

• We can apply enum on switch statement as in the given example:

• **Example of applying Enum on a switch statement**

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURD AY}
public static void main(String args[]){
Day day=Day.MONDAY;
switch(day){
case SUNDAY:
System.out.println("sunday");
break;
case MONDAY:
System.out.println("monday");
break;
default:
System.out.println("other day");
}
}
}
```

• **Output:**

• Monday

1.10 Java - Strings Class

• Strings, which are widely used in Java programming, are a sequence of characters.

• In the Java programming language, strings are objects.

• The Java platform provides the String class to create and manipulate strings.

Creating Strings:

• The most direct way to create a string is to write:

String greeting = "Hello world!";

• Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

• As with any other object, you can create String objects by using the new keyword and a constructor.

• The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters.

• public class StringDemo

```
{
public static void main(String args[])
{
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
String helloString = new String(helloArray); System.out.println(
helloString );
}
}
```

• This would produce the following result:

• hello.

• **Note:** The String class is immutable, so that once it is created a String object cannot be changed.

• If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes

.• **String Length:**

• Methods used to obtain information about an object are known as accessor methods.

• One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

• Below given program is an example of **length()** , method String class.

```
public class StringDemo {
public static void main(String args[])
{
String palindrome = "Dot saw I was Tod";
int len = palindrome.length();
System.out.println( "String Length is : " + len );
}
}
```

• **This would produce the following result:**

• String Length is : 17

• **Concatenating Strings:**

• The String class includes a method for concatenating two strings:

• string1.concat(string2);

• This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

• "My name is ".concat("Zara");

• Strings are more commonly concatenated with the + operator, as in:

• "Hello," + " world" + "!"

• which results in:

• "Hello, world!"

• **Let us look at the following example:**

```
public class StringDemo
{
public static void main(String args[])
{
```

String string1 = "saw I was ";

System.out.println("Dot " + string1 + "Tod");

}

}

• **This would produce the following result:**

• Dot saw I was Tod

• **Creating Format Strings:**

• You have printf() and format() methods to print output with formatted numbers.

• The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

• Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.

For example, instead of:

• System.out.printf("The value of the float variable is " +
"%f, while the value of the integer " +
"variable is %d, and the string " +
"is %s", floatVar, intVar, stringVar);

• **you can write:**

• String fs;fs = String.format("The value of the float variable is " +
"%f, while the value of the integer " + "variable is %d, and
the string " + "is %s", floatVar, intVar, stringVar);

• System.out.println(fs);

String Methods:

Here is the list of methods supported by String class:

SN	Methods with Description
1	**char charAt(int index)** Returns the character at the specified index.
2	**int compareTo(Object o)** Compares this String to another Object.
3	**int compareTo(String anotherString)** Compares two strings lexicographically.
4	**int compareToIgnoreCase(String str)** Compares two strings lexicographically, ignoring case differences.
5	**String concat(String str)** Concatenates the specified string to the end of this string.
6	**boolean contentEquals(StringBuffer sb)** Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	**static String copyValueOf(char[] data)** Returns a String that represents the character sequence in the array specified.

8	**static String copyValueOf(char[] data, int offset, int count)** Returns a String that represents the character sequence in the array specified.
9	**boolean endsWith(String suffix)** Tests if this string ends with the specified suffix.
10	**boolean equals(Object anObject)** Compares this string to the specified object.
11	**boolean equalsIgnoreCase(String anotherString)** Compares this String to another String, ignoring case considerations.
12	**byte getBytes()** Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	**byte[] getBytes(String charsetName** Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	**void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** Copies characters from this string into the destination character array.
15	**int hashCode()** Returns a hash code for this string.
16	**int indexOf(int ch)** Returns the index within this string of the first occurrence of the specified character.
17	**int indexOf(int ch, int fromIndex)** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	**int indexOf(String str)** Returns the index within this string of the first occurrence of the specified substring.

19	**int indexOf(String str, int fromIndex)** Returns the index within this string of the first occurrence of the specified substring, starting at the specified index
20	**String intern()** Returns a canonical representation for the string object.
21	**int lastIndexOf(int ch)** Returns the index within this string of the last occurrence of the specified character.
22	**int lastIndexOf(int ch, int fromIndex)** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	**int lastIndexOf(String str)** Returns the index within this string of the rightmost occurrence of the specified substring.
24	**int lastIndexOf(String str, int fromIndex)** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	**int length()** Returns the length of this string.
26	**boolean matches(String regex)** Tells whether or not this string matches the given regular expression.
27	**boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** Tests if two string regions are equal.
28	**boolean regionMatches(int toffset, String other, int ooffset, int len)** Tests if two string regions are equal
29	**String replace(char oldChar, char newChar)** Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
30	**String replaceAll(String regex, String replacement** Replaces each substring of this string that matches the given regular

	expression with the given replacement.
31	**String replaceFirst(String regex, String replacement)** Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	**String[] split(String regex)** Splits this string around matches of the given regular expression.
33	**String[] split(String regex, int limit)** Splits this string around matches of the given regular expression.
34	**boolean startsWith(String prefix)** Tests if this string starts with the specified prefix.
35	**boolean startsWith(String prefix, int toffset)** Tests if this string starts with the specified prefix beginning a specified index.
36	**CharSequence subSequence(int beginIndex, int endIndex)** Returns a new character sequence that is a subsequence of this sequence.
37	**String substring(int beginIndex)** Returns a new string that is a substring of this string.
38	**String substring(int beginIndex, int endIndex)** Returns a new string that is a substring of this string.
39	**char[] toCharArray()** Converts this string to a new character array.
40	**String toLowerCase()** Converts all of the characters in this String to lower case using the rules of the default locale.
41	**String toLowerCase(Locale locale)** Converts all of the characters in this String to lower case using the rules of the given Locale.
42	**String toString()** This object (which is already a string!) is itself returned.
43	**String toUpperCase()** Converts all of the characters in this String to upper case using the rules of the default locale.

44	**String toUpperCase(Locale locale)** Converts all of the characters in this String to upper case using the rules of the given Locale.
45	**String trim()** Returns a copy of the string, with leading and trailing whitespace omitted.
46	**static String valueOf(primitive data type x)** Returns the string representation of the passed data type argument.

1.11 Java - Inheritance

• Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.

• With the use of inheritance the information is made manageable in a hierarchical order.

• The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

• **extends** is the keyword used to inherit the properties of a class. Below given is the syntax of extends keyword.

```
Class Super
{
..........
}
Class Sub extends Super
{
```

....

}

•

Note − A subclass inherits all the members (fields, methods, and nested classes) from its superclass.

• Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

• **IS-A Relationship**

• IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal
{ }
public class Mammal extends Animal
{
}
public class Reptile extends Animal {
}
public class Dog extends Mammal
{
}
```

• Now, based on the above example, In Object Oriented terms, the following are true −

• Animal is the super class of Mammal class.

• Animal is the super class of Reptile class.

• Mammal and Reptile are subclasses of Animal class.

• Dog is the subclass of both Mammal and Animal classes.• Now, if we consider the IS-A relationship, we can say −

• Mammal IS-A Animal

• Reptile IS-A Animal

• Dog IS-A Mammal

• Hence : Dog IS-A Animal as well•

With use of the extends keyword the subclasses will be able to inherit all the properties of the

super class except for the private properties of the super class.

• We can assure that Mammal is actually an Animal with the use of the instance operator.

• **Example**

```
Class Animal {
}
class Mammal extends Animal
{
}
Class Reptile extends Animal {
}
public class Dog extends Mammal
{
public static void main(String args[])
{
Animal a =newAnimal();
Mammal m =newMammal();
Dog d =newDog();
System.out.println(m instanceofAnimal);
System.out.println(d instanceofMammal);
System.out.println(d instanceofAnimal);
}
}
```

This would produce the following result −

```
true
true
true
```

• Since we have a good understanding of the **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship.

• Generally, the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

• **Example**

```
public interface Animal {
}
public class Mammal implements Animal {
}
public class Dog extends Mammal
{
}
```

• **The instanceof Keyword**

• Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal

```
interface Animal {
}
class Mammal implements Animal
{
}
public class Dog extends Mammal
{
public static void main(String args[])
{
Mammal m =new Mammal();
Dog d =new Dog();
System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);}}
```

• **This would produce the following result:**

```
true
true
true
```

• **HAS-A relationship**

• These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing.

• This relationship helps to reduce duplication of code as well as bugs.

• **Lets us look into an example −**

• publicclassVehicle{}publicclassSpeed{} publicclassVanextendsVehicle

{

privateSpeed sp;}

• This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

• In Object-Oriented feature, the users do not need to bother about which object is doing the real work.

• To achieve this, the Van class hides the implementation details from the users of the Van class.

• So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

• A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal −

• public class extends Animal,Mammal

• {

• }

• However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.A **base class** is a **class**, in an **object**-oriented programming language, from which other **classes** are derived. It facilitates the creation of other **classes** that can reuse the code implicitly inherited from the **base class** (except constructors and destructors).

Base Class

Definition - What does *Base Class* mean?

A base class is a class, in an object-oriented programming language, from which other classes are derived. It facilitates the creation of other classes that can reuse the code implicitly inherited from the base class (except constructors and
destructors). A programmer can extend base class functionality by adding or overriding members relevant to the derived class.

A base class may also be called parent class or superclass.

Techopedia explains *Base Class*

A class derived from a base class inherits both data and behavior. For example, "vehicle" can be a base class from which "car" and "bus" are derived. Cars and buses are both vehicles, but each represents its own specialization of the vehicle base class.

- **A base class has the following properties:**
- Base classes are automatically instantiated before derived classes.
- The derived class can communicate to the base class during instantiation by calling the base class constructor with a matching parameter list.
- Base class members can be accessed from the derived class through an explicit cast.
- If abstract methods are defined in a base class, then this class is considered to be an abstract class and the non-abstract derived class should override these methods.
- Abstract base classes are created using the "abstract" keyword in its declaration and are used to prevent direct initiation using the "new" keyword

Java Inheritance (Subclass and Superclass)

- In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:
- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

- A **Java subclass** is a class which inherits a method or methods from a **Java** superclass.
- A **Java** class may be either a **subclass**, a superclass, both, or neither.

•In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

• **Creating Subclasses**

• You declare that a class is the subclass of another class within the class declaration.

• For example, suppose that you wanted to create a subclass named SubClass of another class named SuperClass. You would write:

• class SubClass extends SuperClass {

• . . .

• }

• This declares that SubClass is the subclass of the Superclass class.

• It also implicitly declares that SuperClass is the superclass of SubClass.

• A subclass also inherits variables and methods from its superclass's superclass, and so on up the inheritance tree.

• A Java class can have only one direct superclass. Java does not support multiple inheritance.

• Creating a subclass can be as simple as including the extends clause in your class declaration.

• However, you usually have to make other provisions in your code when subclassing a class, such as overriding methods or providing implementation for abstract methods.

• **What Member Variables Does a Subclass Inherit?**

• **Rule:** A subclass inherits all of the member variables within its superclass that are accessible to that subclass (unless the member variable is hidden by the subclass).

• **The following list itemizes the member variables that are inherited by a subclass:**

• Subclasses inherit those member variables declared as public or protected.

• Subclasses inherit those member variables declared with no access specifier as long as the subclass is in the same package as the superclass.

• Subclasses don't inherit a superclass's member variable if the subclass declares a member variable using the same name. The subclass's member variable is said to hide the member variable in the superclass.

• Subclasses don't inherit the superclass's private member variables.

• **Hiding Member Variables**

• As mentioned in the previous section, member variables defined in the subclass hide member variables of the same name in the superclass.

• While this feature of the Java language is powerful and convenient, it can be a fruitful source of errors: hiding a member variable can be done deliberately or by accident.

• So, when naming your member variables be careful to only hide those member variables that you actually wish to hide.

• One interesting feature of Java member variables is that a class can access a hidden member variable through its superclass.

• Consider this superclass and subclass pair:

```
class Super {
Number aNumber;
}
class Sub extends Super {
Float aNumber;
}
```

• The aNumber variable in Sub hides aNumber in Super. But you can access aNumber from the superclass with:

• super.aNumber

• super is a Java language keyword that allows a method to refer to hidden variables and overriden methods of the superclass.

• **What Methods Does a Subclass Inherit?**

• The rule that specifies which methods get inherited by a subclass is similar to that for member variables.

• **Rule:** A subclass inherits all of the methods within its superclass that are accessible to that subclass (unless the method is overriden by the subclass).

• The following list itemizes the methods that are inherited by a subclass:

• Subclasses inherit those methods declared as public or protected.

• Subclasses inherit those superclass methods declared with no access specifier as long as the subclass is in the same package as the superclass.

• Subclasses don't inherit a superclass's method if the subclass declares a method using the same name. The method in the subclass is said to override the one in the superclass.

• Subclasses don't inherit the superclass's private methods.

• A subclass can either completely override the implementation for an inherited method, or the subclass can enhance the method by adding functionality to it.

• **Overriding Methods**

• The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is "close enough" and then supplement or modify the behavior of that superclass.

Forms Of Inheritance

Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.

Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.

Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.

Generalization. The child class modifies or overrides some of the methods of the parent class

. ***Extension.*** The child class adds new functionality to the parent class, but does not change any inherited behavior.

Limitation. The child class restricts the use of some of the behavior inherited from the parent class.

Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.

Combination. The child class inherits features from more than one parent class. This is multiple inheritance.

1) Specialization Inheritance

• By far the most common form of inheritance is for specialization.

• A good example is the Java hierarchy of Graphical components in the AWT:

• Component

– Label

– Button

– TextComponent

• TextArea

• TextField

– CheckBox

– ScrollBar

• Each child class overrides a method inherited from the parent in order to specialize the class in some way.

2) Specification Inheritance

• If the parent class is abstract, we often say that it is providing a specification for the child class, and therefore it is specification inheritance (a variety of specialization inheritance).

• Example: Java Event Listeners

• ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

3) Inheritance for Construction

• If the parent class is used as a source for behavior, but the child class has no *is-a* relationship to the parent, then we say the child class is using inheritance for construction.

• An example might be subclassing the idea of a **Set** from an existing **List** class.

• Generally not a good idea, since it can break the principle of substituability, but nevertheless sometimes found in practice.

(More often in dynamically typed languages, such as Smalltalk).

4) Inheritance for Generalization or Extension

• If a child class generalizes or extends the parent class by providing more functionality, but does not override any method, we call it inheritance for generalization.

• The child class doesn't change anything inherited from the parent, it simply adds new features.

• An example is Java Properties inheriting form Hash table.

5) Inheritance for Limitation

• If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it inheritance for limitation.

• For example, you have an existing **List** data type that allows items to be inserted at either end, and you override methods allowing insertion at one end in order to create a **Stack**.

• Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice.

• **6) Inheritance for Variance**

• Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.

• **Example: Mouse and TouchPad and JoyStick**

• Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

• **Benefits of Inheritance**

• Software Reuse

• Code Sharing

• Improved Reliability

• Consistency of Interface

• Rapid Prototyping

• Polymorphism

• Information Hiding

• **Cost of Inheritance**

• Execution speed

• Program size

• Message Passing Overhead

• Program Complexity

• This does not mean you should not use inheritance, but rather than you must understand the benefits, and weigh the benefits against the costs.

➢ **super keyword in java**

• It is used inside a sub-class method definition to call a method defined in the **super**class.

• Private methods of the **super**-class cannot be called. Only public and protected methods can be called by the **super keyword**.

•It is also used by class constructors to invoke constructors of its parent class.

• **Usage** of **java super Keyword**.

• **super** is **used** to refer immediate parent class instance variable.

•**super**() is **used** to invoke immediate parent class constructor.

•**super** is **used** to invoke immediate parent class method.

• The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

• Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

• **Usage of java super Keyword**

1) super is used to refer immediate parent class instance variable.

2) super() is used to invoke immediate parent class constructor.

3) super is used to invoke immediate parent class method.

• **1) super is used to refer immediate parent class instance variable.**

```
//Parent class or Superclass
 class Parentclass
 {
int num=100;
 }
 //Child class or subclass
 class Subclass extends Parent class
 {
 /* I am declaring the same variable * num in child class too. */
 int num=110;
 void printNumber()
 {
 System.out.println(num);
 }
 public static void main(String args[ ])
 {
 Subclass obj=newSubclass( );
 obj.printNumber();
 }
 }
```

• **Sample Output:**

110

• In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is referred by instance by default, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

• ***Solution by super keyword***

• //example of super keyword

```
class Vehicle{
int speed=50;
}
class Bike4 extends Vehicle{
int speed=100;
void display(){
System.out.println(super.speed);//will print speed of Vehicle now
}
public static void main(String args[]){
Bike4 b=new Bike4();
b.display();
}
}
```

• **Sample Output:**

• 50

• In the above program we have the same variable “num” declared in both parent class and sub class.

There is no way we can access the num variable of parent class without using super keyword.

• **Accessing the num variable of parent class:**

```
//Parent class or Superclass
class Parentclass
{
int num=100;
```

```
}
//Child class or subclass
class Subclass extends Parentclass
{
int num=110;
void printNumber()
{
//Super.variable_name
System.out.println(super.num);
}
public static void main(String args[ ])
{
Subclass obj=new Subclass( );
obj.printNumber();
}
}
```

- **Sample Output:**

100

- As you can see by using super.num we accessed the num variable of parent class

2) super() to invoke constructor of parent class

- First we will see compiler default behavior.
- When we create the object of sub class, first the constructor of parent class gets invoked and then the constructor of child class.
- It happens because compiler itself adds super()[it invokes parent class constructor to the constructor of child class.

```
class Parentclass
{
Parentclass( )
{
System.out.println("Constructor of Superclass");
```

```
}
}
class Subclass extends Parentclass
{
Subclass( )
{
/* Compile adds super() here at the first line
* of this constructor implicitly
*/
System.out.println("Constructor of Subclass");
}
Subclass(int num)
{
/* Compile adds super() here at the first line * of this constructor implicitly
*/
System.out.println("Constructor with arg");
}
void display()
{
System.out.println("Hello");
}
public static void main(String args[ ])
{
// Creating object using default constructor
Subclass obj=newSubclass();
//Calling sub class method
obj.display();
//Creating object 2 using arg constructor
Subclass obj2=newSubclass(10);
obj2.display( );
}
```

}

• **Sample Output:**

• Constructor of Superclass

• Constructor of Subclass

• Hello

• Constructor of Superclass

• Constructorwith arg

• Hello

• ***We can call super() explicitly too***

```
class Parentclass
{
Parentclass( )
{
System.out.println("Constructor of Superclass");
} }
class Subclass extends Parentclass
{
Subclass( )
{
/* super() must be added to the first
* line of constructor otherwise it would
* throw compilation error:
* " Constructor call must be the first statement
* in a constructor".
*/
super(); System.out.println("Constructor of Subclass");
} void display( )
{
System.out.println("Hello");
}
public static void main(String args[ ])
```

{ Subclass obj=new Subclass( ); obj.display( );

} }

• **Sample Output:**

• Constructor of Superclass

• Constructor of Subclass

• Hello

• **Note:**

1) super() must be the first statement in constructor otherwise we will get the compilation error message: “Constructor call must be the first statement in a constructor”

2) We can also invoke **parameterized constructor** of parent class by providing arguments while calling super. For e.g. super(10) would invoke the parametrized constructor [having one integer argument] of parent class. Similarly super(“hi”) would invoke constructor having String argument.

• **3) super.<method_name> to invoke parent class method**

• super.method_name calls Overridden method.

•

For e.g. Please refer the comments in the below program to understand better:

```
class Parentclass
{
//Overridden method
void display( )
{
System.out.println("Parent class method");
}
}
class Subclass extends Parentclass
{
//Overriding method
void display( )
{
System.out.println("Child class method");
```

```
}void printMsg( )
{
//This would call Overriding method
display( );
//This would call Overridden method
super.display( );
}
public static void main(String args[ ])
{
Subclass obj=new Subclass() ;
obj.printMsg( );
}
}
```

• **Sample Output:**

• Childclass method

• Parentclass method•

If there is no method overriding then we do not need to use super to call the parent class method.

• Consider this example. Here we are able to call parent class method without using super keyword.

```
class Parentclass
{
void display( )
{
System.out.println("Parent class method");
}
}
class Subclass extends Parentclass
{
void printMsg( )
{
```

```
/* This would call parent class method
* no need to use super keyword
*/
display( );
}
public static void main(String args[ ])
{
Subclass obj=new Subclass( );
obj.printMsg( );
}
}
```

• **Sample Output:**

• Parentclass method

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

• **1) Java final variable**

• If you make any variable as final, you cannot change the value of final variable(It will be constant).

• Example of final variable

• There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
final int speedlimit=90;//final variable
void run(){
speedlimit=400;
}
public static void main(String args[]){
Bike9 obj=new Bike9();
obj.run();
}
}//end of class
```

1.12 •Polymorphism is the ability of an object to take on many forms.

•The most common use of **polymorphism** in OOP occurs when a parent class reference is used to refer to a child class object.

• Any **Java** object that can pass more than one IS-A test is considered to be polymorphic.

What is polymorphism in programming?

• Polymorphism is the capability of a method to do different things based on the object that it is acting upon.

• In other words, polymorphism allows you define one interface and have multiple implementations.

•It is a feature that allows one interface to be used for a general class of actions.

•An operation may exhibit different behavior in different instances.

• The behavior depends on the types of data used in the operation.

• It plays an important role in allowing objects having different internal structures to share the same external interface.

• Polymorphism is extensively used in implementing inheritance.• Following concepts demonstrate different types of polymorphism in java.

1) **Method Overloading**

2) **Method Overriding**

• **Method Definition:**

A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.

• 1)**Method Overloading:**

In Java, it is possible to define two or more methods of same name in a class, provided that there argument list or parameters are different. This concept is known as Method Overloading.

• I have covered method overloading and Overriding below. To know more about polymorphism types refer my post **Types of polymorphism in java: Static, Dynamic, Runtime and Compile time Polymorphism**.

• **1) Method Overloading**

• To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call.

• Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method.

• When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

• It allows the user to achieve compile time polymorphism.

• An overloaded method can throw different exceptions.

• **It can have different access modifiers.**

• **Example:**

```
class Overload
{
void demo (int a)
{
System.out.println ("a: "+ a);
}
void demo (int a,int b)
{
System.out.println ("a and b: "+ a +","+ b);
```

```
}
double demo(double a)
{
System.out.println("double a: "+ a);
return a*a;
}}
class MethodOverloading
{
public static void main (String args [])
{
Overload Obj=new Overload();
double result;
Obj.demo(10);
Obj.demo(10,20);
result=Obj.demo(5.5);
System.out.println("O/P : "+ result);
}
}
```

• Here the method demo() is overloaded 3 times: first having 1 int parameter, second one has 2 int parameters and third one is having double arg. The methods are invoked or called with the same type and number of parameters used.

• **Output:**

• a:10

• aand b:10,20

• double a:5.5

• O/P :30.25

• **Rules for Method Overloading**

• Overloading can take place in the same class or in its sub-class.

• Constructor in Java can be overloaded

• Overloaded methods must have a different argument list.

• Overloaded method should always be the part of the same class(can also take place in sub class), with same name but different parameters.

• The parameters may differ in their type or number, or in both.

• They may have the same or different return types.

• It is also known as compile time polymorphism.

• **2) Method Overriding**

• Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class. This feature is known as method overriding.

Example:

```
public class BaseClass
{
public void methodToOverride()//Base class method
{
System.out.println ("I'm the method of BaseClass");
}
}
public class DerivedClass extends BaseClass
{
public void methodToOverride()//Derived Class method
{
System.out.println ("I'm the method of DerivedClass");
}
}
public class TestMethod
{
public static void main (String args [ ])
{
// BaseClass reference and object
BaseClass obj1 =newBaseClass();
// BaseClass reference but DerivedClass object
```

```
BaseClass obj2 =newDerivedClass();
// Calls the method from BaseClass classobj1.methodToOverride();
//Calls the method from DerivedClass classobj2.methodToOverride();
}
}
```

Output:

- I'm the method of BaseClass
- I'm the method of DerivedClass

- **Rules for Method Overriding:**
- applies only to inherited methods
- object type (NOT reference variable type) determines which overridden method will be used at runtime
- Overriding method can have different return type (**refer this**)
- Overriding method must not have more restrictive access modifier
- Abstract methods must be overridden
- Static and final methods cannot be overridden
- Constructors cannot be overridden
- It is also known as Runtime polymorphism

.•**Using the super Keyword**

- When invoking a superclass version of an overridden method the **super** keyword is used.
-

Example

```
class Animal
{
public void move()
{
System.out.println("Animals can move");
}
}
class Dog extends Animal
```

```
{
public void move()
{
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}public class TestDog
{
public static void main(String args[ ])
{
Animal b =newDog();// Animal reference but Dog object
b.move();// runs the method in Dog class
}
}
```

- This will produce the following result −
- Output
- Animals can move
- Dogs can walk and run•An abstract class is a class that is declared abstract —it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be **subclassed**. When an abstract class is **subclassed**, the **subclass** usually provides implementations for all of the abstract methods in its parent class.

Abstract class in Java

•A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

•Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

•**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

•Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

•Abstraction lets you focus on what the object does instead of how it does it.

• **Ways to achieve Abstaction**

• There are two ways to achieve abstraction in java

• Abstract class (0 to 100%)

• Interface (100%)

• **Abstract class in Java**

• A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

• **Example abstract class**

• **abstract class** A{}

• abstract method

• A method that is declared as abstract and does not have implementation is known as abstract method.

• **Example abstract method**

• **abstract void** print Status();//no body and abstract

• Example of abstract class that has abstract method

• In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
abstract class Bike{
abstract void run();
}
class Honda4 extends Bike{
void run()
{
System.out.println("running safely..");
}
public static void main(String args[])
```

```
{
Bike obj = new Honda4();
obj.run();
}
}
```

Java - Overriding

If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

- **Example**
- Let us look at an example.

```
class Animal
{
public void move()
{
System.out.println("Animals can move");
}
}
class Dog Extends Animal
{
public void move( )
{
System.out.println("Dogs can walk and run");
}
}public class TestDog
{
public static void main(String args[ ])
{
Animal a =new Animal( );
```

```
// Animal reference and object
Animal b =new Dog();
// Animal reference but Dog object
a.move(); // runs the method in Animal class
b.move();// runs the method in Dog class
}
}
```

This will produce the following result −

• **Output**

• Animals can move

• Dogs can walk and run• In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class.

• The reason for this is: In compile time, the check is made on the reference type.

• However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

• Therefore, in the above example, the program will compile properly since Animal class has the method move.

• Then, at the runtime, it runs the method specific for that object.

• **Consider the following example −**

• Example

```
class Animal
{
public void move( )
{
System.out.println("Animals can move");
}
}
class Dog extends Animal
{
public void move()
{
```

```
System.out.println("Dogs can walk and run");
}
public void bark()
{System.out.println("Dogs can bark");
}
}
public class TestDog
{
public static void main(String args[ ])
{
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object
a.move();// runs the method in Animal class
b.move();// runs the method in Dog class
b.bark();
}
}
```

• This will produce the following result −

• **Output**

• TestDog.java:26: error: cannot find symbol b.bark();

• symbol: method bark()

• location: variable b of type Animal1 error

• This program will throw a compile time error since b's reference type

• Animal doesn't have a method by the name of bark.•

Rules for Method Overriding

• The argument list should be exactly the same as that of the overridden method.

• The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

• The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.

• Instance methods can be overridden only if they are inherited by the subclass.

• A method declared final cannot be overridden.

• A method declared static cannot be overridden but can be re-declared.

• If a method cannot be inherited, then it cannot be overridden.

• A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

• A subclass in a different package can only override the non-final methods declared public or protected.

• An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method. • Constructors cannot be overridden.

• **Using the super Keyword**

• When invoking a superclass version of an overridden method the **super** keyword is used.

• **Example**

```
class Animal
{
public void move()
{
System.out.println("Animals can move");
}
}
class Dog extends Animal
{
public void move()
{
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}public class TestDog
```

```
{
public static void main(String args[ ])
{
Animal b =newDog();// Animal reference but Dog object
b.move();// runs the method in Dog class
}
}
```

• This will produce the following result −

• **Output**

• Animals can move

• Dogs can walk and run• Method Overloading in Java

• If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

• If we have to perform only one operation, having same name of the methods increases the readability of the program.

• Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Advantage of method overloading

• Method overloading *increases the readability of the program.*

Different ways to overload the method

• Changing the Number of Parameters.

• Changing Data Types of the Arguments.

• Changing the Order of the Parameters of Methods

• **Note**: In Java, Method Overloading is not possible by changing the return type of the method only.

• **1. Changing the Number of Parameters**

• Method overloading can be achieved by changing the number of parameters while passing to different methods.

```
// Java Program to Illustrate Method Overloading
// By Changing the Number of Parameters
// Importing required classes
import java.io.*;
// Class 1
// Helper class
class Product {
// Method 1
// Multiplying two integer values
public int multiply(int a, int b)
{
int prod = a * b;
return prod;
}
// Method 2
// Multiplying three integer values
public int multiply(int a, int b, int c)
{
•
int prod = a * b * c;
return prod;
}
}// Class 2
// Main class
class GFG {
// Main driver method
public static void main(String[] args)
{
```

```
// Creating object of above class inside main()
// method
Product ob = new Product();
// Calling method to Multiply 2 numbers
int prod1 = ob.multiply(1, 2);
// Printing Product of 2 numbers
System.out.println(
"Product of the two integer value :" + prod1);
// Calling method to multiply 3 numbers
int prod2 = ob.multiply(1, 2, 3);
// Printing product of 3 numbers
System.out.println(
"Product of the three integer value :" + prod2);
}
}
```

•

Output:

- Product of the two integer value :2
- Product of the three integer value :6•

2. Changing Data Types of the Arguments

- In many cases, methods can be considered Overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

```
// Java Program to Illustrate Method Overloading
// By Changing Data Types of the Parameters
// Importing required classes
import java.io.*;
// Class 1
// Helper class
class Product {
// Multiplying three integer values
public int Prod(int a, int b, int c)
```

```
{
int prod1 = a * b * c;
return prod1;
}
// Multiplying three double values.
public double Prod(double a, double b, double c)
{
double prod2 = a * b * c;
return prod2;
}
}class GFG {
public static void main(String[] args)
{
Product obj = new Product();
int prod1 = obj.Prod(1, 2, 3);
System.out.println("Product of the three integer value :" + prod1);
double prod2 = obj.Prod(1.0, 2.0, 3.0);
System.out.println("Product of the three double value :" + prod2);
}
}
```

- **Output:**
- Product of the three integer value :6 Product of the three double value :6.0

•3. Changing the Order of the Parameters of Methods

- Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods.
- For example, if the parameters of method 1 are (String name, int roll_no) and the other method is (int roll_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

```
// Java Program to Illustrate Method Overloading
// By changing the Order of the Parameters
// Importing required classes
import java.io.*;
// Class 1
// Helper class
class Student {
// Method 1
public void StudentId(String name, int roll_no)
{
System.out.println("Name :" + name + " " + "Roll-No :" + roll_no);
}
// Method 2
public void StudentId(int roll_no, String name)
{
// Again printing name and id of person
System.out.println("Roll-No :" + roll_no + " " + "Name :" + name);
}
}// Class 2
// Main class
class GFG {
public static void main(String[] args)
{
// Creating object of above class
Student obj = new Student();
// Passing name and id
// Note: Reversing order
obj.StudentId("Spyd3r", 1);
obj.StudentId(2, "Kamlesh");
}
}
```

• **Output:**

• Name :Spyd3r Roll-No :1

• Roll-No :2 Name :Kamlesh

• **Why Method Overloading is not possible by changing the return type of method only?**

• In java, method overloading is not possible by changing the return type of the method only because of ambiguity(not clear, confusion).

• Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){
return a+b;
}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}
}
```

Output:

Compile Time Error: method add(int,int) is already defined in class Adder static double add(int a,int b){return a+b;}

• **Can we overload java main() method?**

• Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args){System.out.println("main with Str
ing[]");}
```

public static void main(String args){System.out.println("main with Strin
g");}
public static void main(){System.out.println("main without args");} }
Output:
• main with String[]

1.13 Packages and Interfaces

A **java package** is a group of similar types of classes, interfaces and sub-packages.
• Package in java can be categorized in two form, built-in package and user-defined package.
• There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using userdefined packages.

Advantages of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

• **Simple example of java package**
• The **package keyword** is used to create a package in java.
• //save as Simple.java
• **package** mypack;
• **public class** Simple{
• **public static void** main(String args[]){
• System.out.println("Welcome to package");
• }
• }
• **How to compile java package**
• If you are not using any IDE, you need to follow the **syntax** given
below:
• javac -d directory javafilename
• For **example**
• javac -d . Simple.java

• The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

• **How to run java package program**

• You need to use fully qualified name e.g. mypack.Simple etc to run the class.

• **To Compile:** javac -d . Simple.java

• **To Run:** java mypack.Simple

• Output:Welcome to package

• The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.

• The .represents the current folder.

• **How to access package from another package?**

• There are three ways to access the package from outside the package.

• import package.*;

• import package.classname;

• fully qualified name.•

1) Using packagename.*

• If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

• The import keyword is used to make the classes and interface of another package accessible to the current package.

• Example of package that import the packagename.*

• //save by A.java

```
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
```

public static void main(String args[ ]){
A obj = **new** A();
obj.msg();
}
}

• Output:

• Hello

2) Using packagename.classname

• If you import package.classname then only declared class of this package will be accessible.

• Example of package by import package.classname

• //save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
class B{
public static void main(String args[]){
A obj = **new** A();
obj.msg();
}
}

• Output:

• Hello

3) Using fully qualified name

• If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

• It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

• Example of package by import fully qualified name

• //save by A.java

```
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
public static void main(String args[]){
pack.A obj = new pack.A();//using fully qualified name
obj.msg();
}
}
```

• **Output:**

• Hello

• ***Note: If you import a package, subpackages will not be imported.***

• If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

• ***Note: Sequence of the program must be package then import then class.***

• **Subpackage in java**

• Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

• Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc.

• These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on.

• So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

• **Example of Subpackage**

```
package com.javatpoint.core;
class Simple{
public static void main(String args[]){
System.out.println("Hello subpackage");
}
}
```

• **To Compile:** javac -d . Simple.java

• **To Run:** java com.javatpoint.core.Simple

• Output:Hello subpackage• How to send the class file to another directory or drive?

• There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:• //save as Simple.java

```
package mypack;
public class Simple{
public static void main(String args[]){
System.out.println("Welcome to package");
}
}
```

• **To Compile:**

• **e:\sources> javac -d c:\classes Simple.java**

• **To Run:**

• To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

• **e:\sources> set classpath=c:\classes;.;**

• **e:\sources> java mypack.Simple**

• Another way to run this program by -classpath switch of java:

• The -classpath switch can be used with javac and java tool.

• To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

• **e:\sources> java -classpath c:\classes mypack.Simple**

• Output:Welcome to package

• **Ways to load the class files or jar files**

• There are two ways to load the class files temporary and permanent.

• **Temporary**

– By setting the classpath in the command prompt

– By -classpath switch

• **Permanent**

– By setting the classpath in the environment variables

– By creating the jar file, that contains all the class files, and
copying the jar file in the jre/lib/ext folder.

• ***Rule: There can be only one public class in a java source file and it must be saved by the public class name.***

• //save as C.java otherwise Compile Time Error

class A{}

class B{}

public class C{}

• How to put two public classes in a package?

• If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same.

For example:

• //save as A.java

package javatpoint;

public class A{}

• //save as B.java

package javatpoint;

public class B{}

• **What is static import feature of Java5?**

• **Static Import:**

• The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

• **Advantage of static import:**

• Less coding is required if you have access any static member of a class oftenly.

• **Disadvantage of static import:**

• If you overuse the static import feature, it makes the program unreadable and unmaintainable.

• **Simple Example of static import**

```
import static java.lang.System.*;
class StaticImportExample{
public static void main(String args[]){
out.println("Hello");//Now no need of System.out
out.println("Java");
}
}
```

Output:

Hello

Java

• **What is the difference between import and static import?**

• The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification.

• The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

• **What about package class?**

• **Package class**

• The package class provides methods to get information about the specification and implementation of a package.

• It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.•

Example of Package class

• In this example, we are printing the details of java.lang package by invoking the methods of package class.

```
•class PackageInfo{
 public static void main(String args[]){
 Package p=Package.getPackage("java.lang");
 System.out.println("package name: "+p.getName());
 System.out.println("Specification Title: "+p.getSpecificationTitle());
 System.out.println("Specification Vendor: "+p.getSpecificationVendor());
 System.out.println("Specification Version: "+p.getSpecificationVersion());
 System.out.println("Implementaion Title: "+p.getImplementationTitle());
 System.out.println("Implementation Vendor: "+p.getImplementationVendor());
 System.out.println("Implementation Version: "+p.getImplementationVersion());
 System.out.println("Is sealed: "+p.isSealed());
 }
 }
```

• **Output:**

• package name: java.lang

• Specification Title: Java Plateform API Specification

• Specification Vendor: Sun Microsystems, Inc.

• Specification Version: 1.6

• Implemenation Title: Java Runtime Environment

• Implemenation Vendor: Sun Microsystems, Inc.

• Implemenation Version: 1.6.0_30

• IS sealed: false

• **Java – Packages**

• Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

• A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management.

• **Some of the existing packages in Java are:**

• **java.lang** - bundles the fundamental classes

• **java.io** - classes for input , output functions are bundled in this package

• Programmers can define their own packages to bundle group of classes/interfaces, etc.

• It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.

• Since the package creates a new namespace there won't be any name conflicts with names in other packages.

• Using packages, it is easier to provide access control and it is also easier to locate the related classes

• **Creating a package:**

• While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

• The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

• If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

• To compile the Java programs with package statements you have to do use -d option as shown below.

• javac-d **Destination_folder file_name**.java

• Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

• **Example:**

• Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes, interfaces.

• Below given package example contains interface named *animals*:

• /* File name : Animal.java */

package animals;

• interface Animal

• {

• public void eat();

• public void travel();

• }

• Now, let us implement the above interface in the same package *animals*:•

package animals;

•

/* File name : MammalInt.java */

public class MammalInt implements Animal

{

public void eat ()

{

System.out.println("Mammal eats");

}

public void travel(

)

{

System.out.println("Mammal travels");

}

public int noOfLegs(

)

{

```
return0;
}
public static void main(String args[ ])
{
MammalInt m =new MammalInt();
m.eat();
m.travel( );
}
}
```

• Now compile the java files as shown below:

• $ javac -d . Animal.java

• $ javac -d . MammalInt.java• Now a package/folder with the name **animals** will be created in the current directory and these class files will be placed in it as shown
below.

• You can execute the class file with in the package and get the result as shown below.

• $ java animals.MammalInt

• Mammal eats

• Mammal travels

• **The import Keyword:**

• If a class wants to use another class in the same package, the package name does not need to be used.

•

Classes in the same package find each other without any special syntax.

• **Example:**

• Here, a class named Boss is added to the payroll package that already contains Employee.

• The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

• package payroll;

• public class Boss

• {

• public void payEmployee(Employee e)

• {

• e.mailCheck();

• }

• }

• What happens if the Employee class is not in the payroll package?

The Boss class must then use one of the following techniques for referring to a class in a different package.

• The fully qualified name of the class can be used.

• **For example:**

• payroll.Employee

• The package can be imported using the import keyword and the wild card (*).

• **For example:**

• import payroll.*;

• The class itself can be imported using the import keyword.

• **For example:**

• import payroll.Employee;

• **Note:** A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

Java - Interfaces

• An interface is a reference type in Java, it is similar to class, it is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

• Along with abstract methods an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

• Writing an interface is similar to writing a class. But a class describes the attributes and behaviours of an object.

• And an interface contains behaviours that a class implements.

• Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

• An interface can contain any number of methods.

• An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.

• The byte code of an interface appears in a **.class** file.

• Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

• However, an interface is different from a class in several ways, including:

• You cannot instantiate an interface.

• An interface does not contain any constructors.

• All of the methods in an interface are abstract.

• An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

• An interface is not extended by a class; it is implemented by a class.

• An interface can extend multiple interfaces.

• **Declaring Interfaces:**

• The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

• **Example:**

• Below given is an example of an interface:

• /* File name : NameOfInterface.java */

```
import java.lang.*;
//Any number of import statements
public interface NameOfInterface
{
//Any number of final, static fields
//Any number of abstract method declarations\
}
```

• **Interfaces have the following properties:**

• An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.

• Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

• Methods in an interface are implicitly public.

• **Example:**

• /* File name : Animal.java */

```
interface Animal
{
public void eat();
public void travel();
}
```

• **Implementing Interfaces:**

• When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

• A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

• /* File name : MammalInt.java */

```
public class MammalInt implements Animal
{
public void eat()
{
System.out.println("Mammal eats");
}
public void travel()
{
System.out.println("Mammal travels");
}
public int noOfLegs()
{
return 0;
}public static void main(String args[])
{
```

```
MammalInt m = new MammalInt();
m.eat();
m.travel();
}
}
```

• **This would produce the following result:**

• Mammal eats

• Mammal travels

• **When overriding methods defined in interfaces there are several rules to be followed:**

• Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

• The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.

• An implementation class itself can be abstract and if so interface methods need not be implemented.

• **When implementation interfaces there are several rules:**

• A class can implement more than one interface at a time.

• A class can extend only one class, but implement many interfaces.

• An interface can extend another interface, similarly to the way that a class can extend another class.

• **Extending Interfaces:**

• An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

• The following Sports interface is extended by Hockey and Football interfaces.

• //Filename: Sports.java

```
public interface Sports
{
```

public void setHomeTeam(String name);

public void setVisitingTeam(String name);

}

//Filename: Football.java

public interface Football extends Sports

{

public void homeTeamScored(int points);

public void visitingTeamScored(int points);

public void endOfQuarter(int quarter);}

• //Filename: Hockey.java

• public interface Hockey extends Sports

• {

•

public void homeGoalScored();

•

public void visitingGoalScored();

•

public void endOfPeriod(int period);

• public void overtimePeriod(int ot);

• }

• The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods.

• Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

• **Extending Multiple Interfaces:**

• A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

• The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

• **For example, if the Hockey interface extended both Sports and Event, it would be declared as:**

• public interface Hockey extends Sports, Event

• Tagging Interfaces:

• The most common use of extending interfaces occurs when the parent interface does not contain any methods.

• For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

• package java.util;

• public interface EventListener

• {

• }

• An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

• **Creates a common parent:** As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces.

• For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

• **Adds a data type to a class:** This situation is where the term tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

What is Interface?

• The interface is a blueprint that can be used to implement a class. The interface does not contain any concrete methods (methods that have code).

• All the methods of an interface are abstract methods.

• An interface cannot be instantiated. However, classes that implement interfaces can be instantiated.

• Interfaces never contain instance variables but, they can contain public static final variables (i.e., constant class variables)

What Is Abstract Class?

• A class which has the abstract keyword in its declaration is called abstract class.

• Abstract classes should have at least one abstract method. , i.e., methods without a body. It can have multiple concrete methods.

• Abstract classes allow you to create blueprints for concrete classes.

• But the inheriting class should implement the abstract method.

• Abstract classes cannot be instantiated.

• **Important Reasons For Using Interfaces**

• Interfaces are used to achieve abstraction.

• Designed to support dynamic method resolution at run time

• It helps you to achieve loose coupling.

• Allows you to separate the definition of a method from the inheritance hierarchy

• **Important Reasons For Using Abstract Class**

• Abstract classes offer default functionality for the subclasses.

• Provides a template for future specific classes

• Helps you to define a common interface for its subclasses

• Abstract class allows code reusability.

• **Difference between abstract class and interface**

• Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

• But there are many differences between abstract class and interface that are given below.

Class

A class is a blueprint from which individual objects are created. A class can contain any of the following variable types.

Local variables − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables − Class variables are variables declared within a class, outside any method, with the static keyword.

• **Interfaces**

• An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods.

• A class implements an interface, thereby inheriting the abstract methods of the interface.

• Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

• Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object.

• And an interface contains behaviors that a class implements.

1.14 Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors*so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception ?

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

• **Advantages of Exception Handling**

• The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take a scenario:

- statement 1;
- statement 2;
- statement 3;
- statement 4;
- statement 5;//exception occurs
- statement 6;
- statement 7;
- statement 8;
- statement 9;
- statement 10;
- Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.
- **Hierarchy of Java Exception classes**
- **Types of Exception**
- There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:
- Checked Exception
- Unchecked Exception
- Error
- **Difference between checked and unchecked exceptions**

1) Checked Exception

- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException,

SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

• Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

• **Common scenarios where exceptions may occur**

• There are given some scenarios where unchecked exceptions can occur.

They are as follows:

1) Scenario where ArithmeticException occurs

• If we divide any number by zero, there occurs an ArithmeticException.

• **int** a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

• If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

• String s=**null**;

• System.out.println(s.length());//NullPointerException

3) Scenario where NumberFormatException occurs

• The wrong formatting of any value, may occur NumberFormatException.

Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

• String s="abc";

• **int** i=Integer.parseInt(s);//NumberFormatException

4) Scenario where ArrayIndexOutOfBoundsException occurs

• If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

• **int** a[]=**new int**[5];

• a[10]=50; //ArrayIndexOutOfBoundsException• **Advantages of Exception Handling in Java**

• Java provides a sophisticated exception handling mechanism that enables you to detect exceptional conditions in your programs and fix the exceptions as and when they occur. Using exception handling features offers several advantages.

Let's examine these advantages in detail.

• **Provision to Complete Program Execution:**

• One of the important purposes of exception handling in Java is to continue program execution after an *exception is caught and handled*.

• The execution of a Java program does not terminate when an exception occurs. Once the exception is resolved, program execution continues till completion.

• By using well-structured *try, catch, and finally blocks*, you can create programs that fix exceptions and continue execution as if there were no errors. If there is a possibility of more than one exception, you can use multiple catch blocks to handle the different exceptions.

• **Easy Identification of Program Code and Error-Handling Code:**

• The use of try/catch blocks segregates error-handling code and program code making it easier to identify the logical flow of a program. The logic in the program code does not include details of the actions to be performed when an *exception* occurs. Such details are present in the catch blocks.

• Unlike many traditional programming languages that include confusing error reporting and error handling code in between the program code, Java allows you to create well-organized code. Separating error handling and program logic in this way makes it easier to understand and maintain programs in the long run.

• **Propagation of Errors:**

• Java's exception handling mechanism works in such a way that error reports are propagated up the call stack. This is because whenever an exception occurs, Java's runtime environment checks the call stack backwards to identify methods that can catch the exception.

• When a program includes several calls between methods, propagation of exceptions up the call stack ensures that exceptions are caught by the right methods.

• **Meaningful Error Reporting:**

• The exceptions thrown in a Java program are objects of a class. Since the Throwable class overrides the toString() method, you can obtain a description of an exception in the form of a string and display the description using a println() statement.

• **Identifying Error Types:**

• Java provides several super classes and sub classes that group exceptions based on their type. While the super classes like *IOException* provide functionality to handle exceptions of a general

type, sub classes like FileNotFoundException provide functionality to handle specific exception types.

• A method can catch and handle a specific exception type by using a sub class object.

• For example, *FileNotFoundException* is a sub class that only handles a file not found exception. In case a method needs to handle multiple exceptions that are of the same group, you can specify an object of a super class in the method's catch statement.

• For example, IOException is a super class that handles all IO-related exceptions.

• **Java Exception Handling Keywords**

• There are 5 keywords used in java exception handling.

• try

• catch

• finally

• throw

• throws

• **Java try-catch**

• **Java try block**

• Java try block is used to enclose the code that might throw an exception. It must be used within the method.

• Java try block must be followed by either catch or finally block.

• ***Syntax of java try-catch***

• **try**{

• //code that may throw exception

• }**catch**(Exception_class_Name ref){}

• ***Syntax of try-finally block***

• **try**{

• //code that may throw exception

• }**finally**{}

• **Java catch block**

• Java catch block is used to handle the Exception. It must be used after the try block only.

• You can use multiple catch block with a single try.

• Problem without exception handling

• Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1{
public static void main(String args[ ])
{
int data=50/0;//may throw exception
System.out.println("rest of the code...");
}
}
```

• **Output:**

• Exception in thread main java.lang.ArithmeticException:/ by zero As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

• There can be 100 lines of code after exception. So all the code after exception will not be executed.

• **Solution by exception handling**

• Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
public static void main(String args[]){
try{
int data=50/0;
}catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code...");
}
}
```

• **Output:**

• Exception in thread main java.lang.ArithmeticException:/ by zerorest of the code...Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

• **Internal working of java try-catch block•** The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

• Prints out exception description.

• Prints the stack trace (Hierarchy of methods where the exception occurred).

• Causes the program to terminate.

• But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

• Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

• ***Using try and catch***

• Try is used to guard a block of code in which exception may occur. This block of code is called guarded region. A catch statement involves declaring the type of exception you are trying to catch.

If an exception occurs in guarded code, the catch block that follows the try is checked, if the type of exception that occured is listed in the catch block then the exception is handed over to the catch block which then handles it.

• ***Example using Try and catch***

```
class Excp {
public static void main(String args[ ])
{
int a,b,c;
try
{
a=0;
b=10;
c=b/a;
```

```
System.out.println("This line will not be executed");
}
catch(ArithmeticException e)
{
System.out.println("Divided by zero");
}
System.out.println("After exception is handled");
}
}
```

- **Output :**
- Divided by zero
- After exception is handled
- An exception will thrown by this program as we are trying to divide a number by zero inside **try** block.
- The program control is transfered outside **try** block
- . Thus the line "*This line will not be executed*" is never parsed by the compiler.
- The exception thrown is handle in **catch** block.
- Once the exception is handled the program controls continue with the next line in the program.
- Thus the line "*After exception is handled*" is printed.• ***Multiple catch blocks:***
- A try block can be followed by multiple catch blocks. You can have any number of catch blocks after a single try block.
- If an exception occurs in the guarded code the exception is passed to the first catch block in the list.
- If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block.
- This continue until the exception is caught or falls through all catches.
- If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.
- **Let's see a simple example of java multi-catch block.**

```
public class TestMultipleCatchBlock{
public static void main(String args[]){
```

```
try{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("task1 is completed");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 completed");
}
catch(Exception e)
{ System.out.println("common task completed");
}
System.out.println("rest of the code...");
}
}
```

• **Output:**

task1 completed rest of the code...

• Rule: At a time only one Exception is occured and at a time only one catch block is executed.

• Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
class TestMultipleCatchBlock1{
public static void main(String args[]){
try{
int a[]=new int[5];
a[5]=30/0;
}
```

```
catch(Exception e)
{
System.out.println("common task completed");
}catch(ArithmeticException e)
{
System.out.println("task1 is completed");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("task 2 completed");
}
System.out.println("rest of the code...");
}
}
```

- **Output:**
- Compile-time error
- ***Example for Unreachable Catch block***
- While using multiple **catch** statements, it is important to remember that exception sub classes inside **catch**must come before any of their super classes otherwise it will lead to compile time error.

```
class Excep {
public static void main(String[ ] args)
{
try
{
int arr[ ]={1,2};
arr[2]=3/0;
}
catch(Exception e) //This block handles all Exception
{
System.out.println("Generic exception");
```

```
}
catch(ArrayIndexOutOfBoundsException e) //This block is unreachable
{
System.out.println("array index out of bound exception");
}
}
}
```

Java Nested try block

- The try block within a try block is known as nested try block in java.
- **Why use nested try block**
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.
- Syntax:
- ....

```
try
{
statement 1;
statement 2;
try
{
statement 1;
statement 2;
}
catch(Exception e)
{
}
}
catch(Exception e)
{
}
```

....

• **Java nested try example**

• **Let's see a simple example of java nested try block.**

```
class Excep6{
public static void main(String args[ ]){
try{
try{
System.out.println("going to divide");
int b =39/0;
} catch(ArithmeticException e)
{ System.out.println(e);}
try{
int a[]=new int[5];
a[5]=4;
}
catch(ArrayIndexOutOfBoundsException e) { System.out.println(e); }
System.out.println("other statement);
}catch(Exception e){System.out.println("handeled");}
System.out.println("normal flow..");
}
}
```

• ***Important points to Remember***

• If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.

• Super class **Throwable** overrides **toString()** function, to display error message in form of string.

• While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.

• In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.

• Only the object of Throwable class or its subclasses can be thrown.

• **Java throw exception**

• **Java throw keyword**

• The Java throw keyword is used to explicitly throw an exception.

• We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

• **The syntax of java throw keyword is given below.**

• **throw** exception;

• Let's see the example of throw IOException.

• **throw new** IOException("sorry device error);

• **Java throw keyword example**

• In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{
static void validate(int age){
if(age<18)
throw new ArithmeticException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[]){
validate(13);
System.out.println("rest of the code...");
}
}
```

• **Output:**

• Exception in thread main java.lang.ArithmeticException:not valid

• throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

• **Syntax :**

• **throw** *ThrowableInstance*

• ***Creating Instance of Throwable class***

• There are two possible ways to get an instance of class Throwable,

• Using a parameter in catch block.

• Creating instance with **new** operator.

• **new** NullPointerException("test"); This constructs an instance of NullPointerException with name test.

• ***Example demonstrating throw Keyword***

```
class Test {
static void avg()
{
try
{
throw new ArithmeticException("demo");
}
catch(ArithmeticException e)
{
System.out.println("Exception caught");
}
}
public static void main(String args[])
{
avg();
}
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

• **Java throws keyword**

• The **Java throws keyword** is used to declare an exception.

• It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

• Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

• Syntax of java throws

• return_type method_name() **throws** exception_class_name{

• //method code

• }

• Which exception should be declared

• **Ans)** checked exception only, because:

• **unchecked Exception:** under your control so correct your code.

• **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

• **Advantages of Java throws keyword**

• Now Checked Exception can be propagated (forwarded in call stack).

• It provides information to the caller of the method about the exception.

• Java throws example

• Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
• import java.io.IOException;
class Testthrows1{
void m()throws IOException{
throw new IOException("device error");//checked exception
}
void n()throws IOException{
```

```
m();
}
void p(){
try{
n();
}catch(Exception e)
{
System.out.println("exception handled");
}
}
public static void main(String args[]){
Testthrows1 obj=new Testthrows1();
obj.p();
System.out.println("normal flow...");
}
}
```

- **Output:**
- exception handlednormal flow...
- Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.
- **There are two cases:**
- Case1:You caught the exception i.e. handle the exception using try/catch.
- Case2:You declare the exception i.e. specifying throws with the method.
- **Case1: You handle the exception**
- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
import java.io.*;
class M{
void method( )throws IOException{
throw new IOException("device error");
}
```

```
}
public class Testthrows2{
public static void main(String args[]){
try{M m=new M();
m.method();
}catch(Exception e){System.out.println("exception handled");}
System.out.println("normal flow...");
}
```

- **Output:**
- exception handled
- normal flow...
- **Case2**: **You declare the exception**
- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.
- ***A)Program if exception does not occur***

```
import java.io.*;
class M{
void method()throws IOException{
System.out.println("device operation performed");
}
}
class Testthrows3{
public static void main(String args[])throws IOException{//declare ex
ception
M m=new M();
m.method();
System.out.println("normal flow...");
}
}
```

• **Output:**

• device operation performed

• normal flow...

• ***B)Program if exception occurs***

```
import java.io.*;
class M{
void method()throws IOException{
throw new IOException("device error");
}
}
class Testthrows4{
public static void main(String args[ ])
throws IOException{//declare exception
M m=new M();
m.method();
System.out.println("normal flow...");
}
}
```

• **Output:**

• Runtime Exception

• **Difference between throw and throws**

1) Java throw keyword is used to explicitly throw an exception.

Java throws keyword is used to declare an exception.

2) Checked exception cannot be propagated using throw only.

Checked exception can be propagated with throws.

3) Throw is followed by an instance.

Throws is followed by class.

4) Throw is used within the method.

Throws is used with the method signature.

5) You cannot throw multiple exceptions.

You can declare Multiple exceptions e.g.

public void method()throws

IOException,SQLException.

• **Java finally block**

• **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.

• Java finally block is always executed whether exception is handled or not.Java finally block follows try or catch block.

• ***Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).•***

Why use java finally

• Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

• Usage of Java finally

• Let's see the different cases where java finally block can be used.

• **Case 1**

• Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock{
public static void main(String args[ ]){
try{
int data=25/5;
System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally { System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
}
}
```

• **Output**:

• 5

• finally block is always executed

• rest of the code...

• **Case 2**

• Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{
public static void main(String args[]){
try{
int data=25/0;
System.out.println(data);
}
catch(NullPointerException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
}
}
```

• **Output**:

• finally block is always executed

• Exception in thread main java.lang.ArithmeticException:/ by zero

• **Case 3**

• Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{
public static void main(String args[]){
try{
int data=25/0;
System.out.println(data);
}
catch(ArithmeticException e){System.out.println(e);}
finally{System.out.println("finally block is always executed");}
System.out.println("rest of the code...");
}
```

}

• **Output**:

• Exception in thread main java.lang.ArithmeticException:/ by zero

finally block is always executed

• the code...

• **Difference between final, finally and finalize.**

- **final**
- **finally**
- **finalize**

1) Final is used to apply restrictions on class, method and variable.

Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

Finally is used to place important code, it will be executed whether exception is handled or not.

Finalize is used to perform clean up processing just before object is garbage collected.

2) Final is a keyword. Finally is a block. Finalize is a method.

• **Java final example**

```
class FinalExample{
public static void main(String[ ] args){
final int x=100;
x=200;//Compile Time Error
}
}
```

• **Java finally example**

```
class FinallyExample{
public static void main(String[] args){
try{
int x=300;
} catch(Exception e){System.out.println(e);}
finally{System.out.println("finally block is executed");}
}
```

```
}
```

• **Java finalize example**

```
class FinalizeExample{
public void finalize( ){System.out.println("finalize called");}
public static void main(String[ ] args){
FinalizeExample f1=new FinalizeExample( );
FinalizeExample f2=new FinalizeExample( );
f1=null;
f2=null;
System.gc( );
}
}
```

Java - Built-in Exceptions

• Java defines several exception classes inside the standard package **java.lang**.

• The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

• Java defines several other types of exceptions that relate to its various class libraries.

• Following is the list of Java Unchecked RuntimeException.

Exception Description

ArithmeticException

Arithmetic error, such as divide-by-zero.

ArrayIndexOutOfBoundsException

Array index is out-of-bounds.

ArrayStoreException

Assignment to an array element of an incompatible type.

ClassCastException

Invalid cast.

IllegalArgumentException

Illegal argument used to invoke a method.

IllegalMonitorStateException

Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException

Environment or application is in incorrect state.

IllegalThreadStateException

Requested operation not compatible with current thread state.

IndexOutOfBoundsException

Some type of index is out-of-bounds.

NegativeArraySizeException

Array created with a negative size.

NullPointerException

Invalid use of a null reference.

NumberFormatException

Invalid conversion of a string to a numeric format.

SecurityException

Attempt to violate security.

StringIndexOutOfBounds

Attempt to index outside the bounds of a string.

UnsupportedOperationException

An unsupported operation was encountered.

• Following is the list of Java Checked Exceptions Defined in java.lang.

Exception Description

ClassNotFoundException

Class not found.

CloneNotSupportedException Attempt to clone an object that does not implement the Cloneable interface.

IllegalAccessException

Access to a class is denied.

InstantiationException

Attempt to create an object of an abstract class or

interface.

InterruptedException

One thread has been interrupted by another

thread.

NoSuchFieldException

A requested field does not exist.

NoSuchMethodException

A requested method does not exist.

Creating Your Own Exception Subclasses

• You can create your own exception class by defining a subclass of Exception.

• The Exception class does not define any methods of its own. It inherits methods provided by Throwable.

• All exceptions have the methods defined by Throwable available to them.

• **They are shown in the following list**.

✓ **Throwable fillInStackTrace( )**

Returns a Throwable object that contains a completed stack trace.

✓ **Throwable getCause( )**

Returns the exception that underlies the current exception.

✓ **String getLocalizedMessage( )**

Returns a localized description.

✓ **String getMessage( )**

Returns a description of the exception.

✓ **StackTraceElement[ ] getStackTrace( )**

Returns an array that contains the stack trace.

✓ **Throwable initCause(Throwable causeExc)**

Associates causeExc with the invoking exception as a cause of the invoking

exception.✓ **void printStackTrace( )**

• Displays the stack trace.

✓ **void printStackTrace(PrintStream stream)**

• Sends the stack trace to the stream.

✓ **void printStackTrace(PrintWriter stream)**

• Sends the stack trace to the stream.

✓ **void setStackTrace(StackTraceElement elements[ ])**

• Sets the stack trace to the elements passed in elements.

✓ **String toString( )**

• Returns a String object containing a description of the exception•

The following program creates a custom exception type.

```
class NumberRangeException extends Exception
{
String msg;
NumberRangeException( )
{
msg = new String("Enter a number between 20 and 100");
}
}
public class MyException
{
public static void main (String args [ ])
{
try
{
int x = 10;
if (x < 20 || x >100) throw new NumberRangeException( );
}
catch (NumberRangeException e)
{
System.out.println (e);
} } }
```

• **Sample Output:**

• **C:\kavitha\java>javac MyException.java**

• **C:\kavitha\java>java MyException**

• **NumberRangeException**

UNIT-II

HTML COMMON TAGS: Introduction, HTML Basics- Text, Colors, Links, Images, Lists, Forms, Frames, Tables, Web Page Design, HTML5, Cascading Style Sheets, Introduction to CSS, Types of CSS.

XML: Introduction, Document Type Definition, XML Schema, Document Object Model, Presenting XML(XSL), XML Parsers: DOM and SAX.

2.1 What is HTML?

- HTML stands for Hyper Text Markup Language .
- HTML is the standard markup language for creating Web pages
- HTML describes the structure of a Web page
- HTML consists of a series of elements
- HTML elements tell the browser how to display the content
- HTML elements label pieces of content such as "this is a heading", "this is a paragraph", "this is a link", etc.

Tag	Description
<!--...-->	Defines a comment
<!DOCTYPE>	Defines the document type
<a>	Defines a hyperlink
<abbr>	Defines an abbreviation or an acronym
<acronym>	Not supported in HTML5. Use <abbr> instead. Defines an acronym
<address>	Defines contact information for the author/owner of a document
<applet>	Not supported in HTML5. Use <embed> or <object> instead. Defines an embedded applet
<area>	Defines an area inside an image map

<aside>	Defines content aside from the page content
<audio>	Defines embedded sound content
<b>	Defines bold text
<base>	Specifies the base URL/target for all relative URLs in a document
<basefont>	Not supported in HTML5. Use CSS instead. Specifies a default color, size, and font for all text in a document
<bdi>	Isolates a part of text that might be formatted in a different direction from other text outside it
<bdo>	Overrides the current text direction
<big>	Not supported in HTML5. Use CSS instead. Defines big text

<blockquote>	Defines a section that is quoted from another source
<body>	Defines the document's body
 	Defines a single line break
<button>	Defines a clickable button
<canvas>	Used to draw graphics, on the fly, via scripting (usually JavaScript)
<caption>	Defines a table caption
<center>	Not supported in HTML5. Use CSS instead. Defines centered text
<cite>	Defines the title of a work
<code>	Defines a piece of computer code

<col>	Specifies column properties for each column within a <colgroup> element
<colgroup>	Specifies a group of one or more columns in a table for formatting
<data>	Adds a machine-readable translation of a given content
<datalist>	Specifies a list of pre-defined options for input controls
<dd>	Defines a description/value of a term in a description list
<del>	Defines text that has been deleted from a document
<details>	Defines additional details that the user can view or hide
<dfn>	Specifies a term that is going to be defined within the content

<dir>	Not supported in HTML5. Use instead. Defines a directory list
<div>	Defines a section in a document
<dl>	Defines a description list
<dt>	Defines a term/name in a description list
<em>	Defines emphasized text
<embed>	Defines a container for an external application
<fieldset>	Groups related elements in a form
<figcaption>	Defines a caption for a <figure> element
<figure>	Specifies self-contained content
<font>	Not supported in HTML5. Use CSS instead. Defines font, color, and size for text

<footer>	Defines a footer for a document or section
<form>	Defines an HTML form for user input
<frame>	Not supported in HTML5. Defines a window (a frame) in a frameset
<frameset>	Not supported in HTML5. Defines a set of frames
<h1> to <h6>	Defines HTML headings
<head>	Contains metadata/information for the document
<header>	Defines a header for a document or section
<hr>	Defines a thematic change in the content
<html>	Defines the root of an HTML document
<i>	Defines a part of text in an alternate voice or mood

<iframe>	Defines an inline frame
<img>	Defines an image
<input>	Defines an input control
<ins>	Defines a text that has been inserted into a document
<kbd>	Defines keyboard input
<label>	Defines a label for an <input> element
<legend>	Defines a caption for a <fieldset> element
	Defines a list item
<link>	Defines the relationship between a document and an external resource (most used to link to style sheets)

<main>	Specifies the main content of a document
<map>	Defines an image map
<mark>	Defines marked/highlighted text
<meta>	Defines metadata about an HTML document
<meter>	Defines a scalar measurement within a known range (a gauge)
<nav>	Defines navigation links
<noframes>	Not supported in HTML5. Defines an alternate content for users that do not support frames
<noscript>	Defines an alternate content for users that do not support client-side scripts
<object>	Defines a container for an external application
<optgroup>	Defines a group of related options in a drop-down list
<option>	Defines an option in a drop-down list
<output>	Defines the result of a calculation
<p>	Defines a paragraph
<param>	Defines a parameter for an object
<picture>	Defines a container for multiple image resources
<pre>	Defines preformatted text
<progress>	Represents the progress of a task
<q>	Defines a short quotation

<rp>	Defines what to show in browsers that do not support ruby annotations
<rt>	Defines an explanation/pronunciation of characters (for East Asian typography)
<ruby>	Defines a ruby annotation (for East Asian typography)
<s>	Defines text that is no longer correct
<samp>	Defines sample output from a computer program
<script>	Defines a client-side script
<section>	Defines a section in a document
<select>	Defines a drop-down list
<small>	Defines smaller text
<source>	Defines multiple media resources for media elements (<video> and <audio>)
<span>	Defines a section in a document
<strike>	Not supported in HTML5. Use <del> or <s> instead. Defines strikethrough text
<strong>	Defines important text
<style>	Defines style information for a document
<sub>	Defines subscripted text
<summary>	Defines a visible heading for a <details> element
<sup>	Defines superscripted text
<svg>	Defines a container for SVG graphics
<table>	Defines a table

<tbody>	Groups the body content in a table
<td>	Defines a cell in a table
<template>	Defines a container for content that should be hidden when the page loads
<textarea>	Defines a multiline input control (text area)
<tfoot>	Groups the footer content in a table
<th>	Defines a header cell in a table
<thead>	Groups the header content in a table
<time>	Defines a specific time (or datetime)
<title>	Defines a title for the document
<tr>	Defines a row in a table

<track>	Defines text tracks for media elements (<video> and <audio>)
<tt>	Not supported in HTML5. Use CSS instead. Defines teletype text
<u>	Defines some text that is unarticulated and styled differently from normal text
	Defines an unordered list
<var>	Defines a variable
<video>	Defines embedded video content
<wbr>	Defines a possible line-break

```
• <!DOCTYPE html>
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph.</p>
</body>
</html>
```

• The <!DOCTYPE html> declaration defines that this document is an HTML5 document

• The <html> element is the root element of an HTML page

• The <head> element contains meta information about the HTML page

• The <title> element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)

• The <body> element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.

• The <h1> element defines a large heading

• The <p> element defines a paragraph

• What is an HTML Element?

- An HTML element is defined by a start tag, some content, and an end tag:

<tagname>Content goes here...</tagname>

- The HTML element is everything from the start tag to the end tag:
- <h1>My First Heading</h1>
- <p>My first paragraph.</p>
- Start tag Element content End tag
- <h1> My First Heading </h1>
- <p> My first paragraph </p>
-
 none *none*
- Note: Some HTML elements have no content (like the <br> element). These elements are called empty elements. Empty elements do not have an end tag!

• **Note:** Some HTML elements have no content (like the <br> element).

These elements are called empty elements. Empty elements do not have an end tag!

- **Nested HTML Elements**

• HTML elements can be nested (this means that elements can contain other elements).

• All HTML documents consist of nested HTML elements.

• The following example contains four HTML elements (<html>, <body>, <h1> and <p>):

• **Example**

• <!DOCTYPE html>

<html>

<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>

</html>

• Sample Output:

• **My First Heading**

• My first paragraph.

• **Example Explained**

• The <html> element is the root element and it defines the whole HTML document.

• It has a start tag <html> and an end tag </html>.

• Then, inside the <html> element there is a <body> element:

<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>

• The <body> element defines the document's body.

• It has a start tag <body> and an end tag </body>.

• Then, inside the <body> element there is two other elements: <h1> and <p>:

• <h1>My First Heading</h1>

<p>My first paragraph.</p>

• The <h1> element defines a heading.

• It has a start tag <h1> and an end tag </h1>:

• <h1>My First Heading</h1>

• The <p> element defines a paragraph.

• It has a start tag <p> and an end tag </p>:

• <p>My first paragraph.</p>

HTML Tag Reference

W3Schools' tag reference contains additional information about these tags and their attributes.

Tag	Description
<html>	Defines the root of an HTML document
<body>	Defines the document's body
<h1> to <h6>	Defines HTML headings

2.2 HTML Attributes

• All HTML elements can have **attributes**

• Attributes provide **additional information** about elements

• Attributes are always specified in **the start tag**

• Attributes usually come in name/value pairs like: **name="value".**

• The href Attribute

• The <a> tag defines a hyperlink. The href attribute specifies the URL of the page the link goes to:

• <!DOCTYPE html>

• <html>

• <body>

• <h2>The href Attribute</h2>

• <p>HTML links are defined with the a tag. The link address is specified in the href attribute:</p>

• Visit W3Schools

• </body>

• </html>

• **Sample Output**

• **The href Attribute**

• HTML links are defined with the a tag. The link address is specified in the href attribute:

• Visit W3Schools

• **The src Attribute**

• The <img> tag is used to embed an image in an HTML page. The src attribute specifies the path to the image to be displayed:

• **Example**

• <img src="img_girl.jpg">• **The width and height Attributes**

• The <img> tag should also contain the width and height attributes, which specifies the width and height of the image (in pixels):

• **Example**

• <img src="img_girl.jpg" width="500" height=" 600">

The src Attribute

HTML images are defined with the img tag, and the filename of the image source is specified in the src attribute. The width and height attributes of the img tag,
defines the width and height of the image:

• **The alt Attribute**

• The required alt attribute for the <img> tag specifies an alternate text for an image, if the image for some reason cannot be displayed. This can be due to slow connection, or an error in the src attribute, or if the user uses a screen reader.

• **Example**

• <img src="img_girl.jpg" alt="Girl with a jacket">

•• <!DOCTYPE html>

• <html>

• <body>

• <h2>The alt Attribute</h2>

• <p>The alt attribute should reflect the image content, so users who cannot see the image gets an understanding of what the image contains:</p>

• <img src="img_girl.jpg" alt="Girl with a jacket" width="500" height="600">

• </body>

• </html>

The alt Attribute

The alt attribute should reflect the image content, so users who cannot see the image gets an understanding of what the image contains:

• **Example**

• See what happens if we try to display an image that does not exist:

• <img src="img_typo.jpg" alt="Girl with a jacket">

• <!DOCTYPE html> <html> <body>

• <img src="img_typo.jpg" alt="Girl with a jacket">

• <p>If we try to display an image that does not exist, the value of the alt attribute will be displayed instead. </p>

• </body>

• </html>

Sample Output

If we try to display an image that does not exist, the value of the alt attribute will be displayed instead.

• **The style Attribute**

• The style attribute is used to add styles to an element, such as color, font, size, and more.

• Example

• <p style="color:red;">This is a red paragraph.</p>

• <!DOCTYPE html>

• <html>

• <body>

• <h2>The style Attribute</h2>

• <p>The style attribute is used to add styles to an element, such as color:</p>

• <p style="color:red;">This is a red paragraph.</p>

• </body>
• </html>

• **Sample Output:**
• **The style Attribute**
• The style attribute is used to add styles to an element, such as color:
• This is a red paragraph.

• **The lang Attribute**
• You should always include the lang attribute inside the <html> tag, to declare the language of the Web page. This is meant to assist search engines and
browsers.
• The following example specifies English as the language:
• <!DOCTYPE html>
<html lang="en">
<body>
...
</body>
</html>
• Country codes can also be added to the language code in the lang attribute. So, the first two characters define the language of the HTML page, and the last two characters define the country.
• The following example specifies English as the language and United States as the country:
• <!DOCTYPE html>
<html lang="en-US">
<body>
...
</body>
</html>

• **The title Attribute**

• The title attribute defines some extra information about an element.

• The value of the title attribute will be displayed as a tooltip when you mouse over the element:

• Example

• <p title="I'm a tooltip">This is a paragraph.</p>• <!DOCTYPE html>

• <html>

• <body>

• <h2 title="I'm a header">The title Attribute</h2>

• <p title="I'm a tooltip">Mouse over this paragraph, to display the title attribute as a tooltip.</p>

• </body>

• </html>

• Sample Output:

• **The title Attribute**

• Mouse over this paragraph, to display the title attribute as a tooltip.

• **NOTE:**

• Always Use Lowercase Attributes • The HTML standard does not require lowercase attribute names.

• The title attribute (and all other attributes) can be written with uppercase or lowercase like **title** or **TITLE**.

• However, W3C **recommends** lowercase attributes in HTML, and **demands** lowercase attributes for stricter document types like XHTML.

2.3 HTML Headings

• HTML headings are titles or subtitles that you want to display on a webpage.

Example

• Heading 1

• Heading 2

• Heading 3

• Heading 4

• Heading 5

• Heading 6

•**Note:** Browsers automatically add some white space (a margin) before and after a heading.• <!DOCTYPE html>

• <html> <body>

• <h1>Heading 1</h1>

• <h2>Heading 2</h2>

• <h3>Heading 3</h3>

• <h4>Heading 4</h4>

• <h5>Heading 5</h5>

• <h6>Heading 6</h6>

• </body>

• </html>

Sample Output

Heading 1

Heading 2

Heading 3

Heading 4

Heading 5

Heading 6

- **Headings Are Important**

• Search engines use the headings to index the structure and content of your web pages.

• Users often skim a page by its headings. It is important to use headings to show the document structure.

• <h1> headings should be used for main headings, followed by <h2> headings, then the less important <h3>, and so on.

• **Note:** Use HTML headings for headings only.

Don't use headings to make text **BIG** or **bold**.

• **Bigger Headings**

• Each HTML heading has a default size. However, you can specify the size for any heading with the style attribute, using the CSS font-size property:

• Example

• <h1 style="font-size:60px;">Heading 1</h1>• <!DOCTYPE html>

• <html>

• <body>

• <h1 style="font-size:60px;">Heading 1</h1>

• <p>You can change the size of a heading with the style attribute, using the font-size property.</p>

• </body>

• </html>

Sample Output:

Heading 1

You can change the size of a heading with the style attribute, using the font-size property.

HTML Tag Reference

W3Schools' tag reference contains additional information about these tags and their attributes.

Tag	Description
<html>	Defines the root of an HTML document
<body>	Defines the document's body
<h1> to <h6>	Defines HTML headings

2.4 HTML Paragraphs

A paragraph always starts on a new line, and is usually a block of text.

HTML Paragraphs

The HTML <p> element defines a paragraph. A paragraph always starts on a new line, and browsers automatically add some white space (a margin) before and after a paragraph.

Example

<p>This is a paragraph.</p>

<p>This is another paragraph.</p>• <!DOCTYPE html>

- <html>
- <body>
- <p>This is a paragraph.</p>
- <p>This is a paragraph.</p>
- <p>This is a paragraph.</p>
- </body>

• </html>

• **Sample Output:**

• This is a paragraph.

• This is a paragraph.

• This is a paragraph.

- ## HTML Display

• You cannot be sure how HTML will be displayed.

• Large or small screens, and resized windows will create different results.

• With HTML, you cannot change the display by adding extra spaces or extra lines in your HTML code.

• The browser will automatically remove any extra spaces and lines when the page is displayed:

• <!DOCTYPE html>

• <html> <body>

• <p> This paragraph contains a lot of lines in the source code, but the browser ignores it.

• </p>

• <p> This paragraph contains a lot of spaces in the source code, but the browser ignores it.

• </p>

• <p> The number of lines in a paragraph depends on the size of the browser window. If you resize the browser window, the number of lines in this paragraph will change.

• </p>

• </body>

• </html>

Sample Output

This paragraph contains a lot of lines in the source code, but the browser ignores it.

This paragraph contains a lot of spaces in the source code, but the browser ignores it.

The number of lines in a paragraph depends on the size of the browser window. If you resize the browser window, the number of lines in this paragraph will change.

- **HTML Horizontal Rules**

• The <hr> tag defines a thematic break in an HTML page, and is most often displayed as a horizontal rule.

• The <hr> element is used to separate content (or define a change) in an HTML page:

```
• <!DOCTYPE html>
• <html>
• <body>
• <h1>This is heading 1</h1>
• <p>This is some text.</p>
• <hr>
• <h2>This is heading 2</h2>
• <p>This is some other text.</p>
• <hr>
• <h2>This is heading 2</h2>
• <p>This is some other text.</p>
• </body>  </html>
```

Sample Output:

This is heading 1

This is some text.

This is heading 2

This is some other text.

This is heading 2

This is some other text.

- **HTML Line Breaks**
- The HTML <br> element defines a line break.
- Use <br> if you want a line break (a new line) without starting a new paragraph:
- Example
- <!DOCTYPE html>
- <html>
- <body>
- <p>This is
a paragraph
with line breaks.</p>
- </body>
- </html>

Sample Output:

This is
a paragraph
with line breaks.

• The HTML <pre> Element

• The HTML <pre> element defines preformatted text.

• The text inside a <pre> element is displayed in a fixed-width font (usually Courier), and it preserves both spaces and line breaks:

• Example:

• <!DOCTYPE html>

• <html>

• <body>

• <p>The pre tag preserves both spaces and line breaks:</p>

• <pre>

• My Bonnie lies over the ocean.

• My Bonnie lies over the sea.

• My Bonnie lies over the ocean.

• Oh, bring back my Bonnie to me.

• </pre>

• </body>

• </html>

Sample Output

The pre tag preserves both spaces and line breaks:

```
My Bonnie lies over the ocean.

My Bonnie lies over the sea.

My Bonnie lies over the ocean.

Oh, bring back my Bonnie to me.
```

HTML Tag Reference

W3Schools' tag reference contains additional information about HTML elements and their attributes.

Tag	Description
<p>	Defines a paragraph
<hr>	Defines a thematic change in the content
 	Inserts a single line break
<pre>	Defines pre-formatted text

2.5 HTML Styles

The HTML style attribute is used to add styles to an element, such as color, font, size, and more.

Example:

```
<!DOCTYPE html>
<html>
<body>
<p>I am normal</p>
<p style="color:red;">I am red</p>
<p style="color:blue;">I am blue</p>
<p style="font-size:50px;">I am big</p>
```

</body>

</html>

Sample Output

I am normal

I am red

I am blue

I am big

• The HTML Style Attribute

• Setting the style of an HTML element, can be done with the style attribute.

• The HTML style attribute has the following syntax:

• <*tagname* style="*property*:*value;*">

• The ***property*** is a CSS property. The ***value*** is a CSS value.

• **Syntax**

• Set the background color for a page to powderblue:

• <body style="backgroundcolor:powderblue;">

<h1>This is a heading</h1>

<p>This is a paragraph.</p>

</body>

• **Example 1:**

• <!DOCTYPE html>

• <html>

• <body style="background-color:powderblue;">

• <h1>This is a heading</h1>

• <p>This is a paragraph.</p>

• </body>

• </html>

Sample Output

This is a heading

This is a paragraph.

• **Example 2:**

• <!DOCTYPE html>

• <html>

• <body>

• <h1 style="background-color:powderblue;">This is a heading</h1>

• <p style="background-color:tomato;">This is a paragraph.</p>

• </body>

• </html>

Sample Output

This is a heading

This is a paragraph.

- **Text Color**
- The CSS color property defines the text color for an HTML element:
- **Example**
- <h1 style="color:blue;">This is a heading</h1>

<p style="color:red;">This is a paragraph.</p>

- <!DOCTYPE html>
- <html>
- <body>
- <h1 style="color:blue;">This is a heading</h1>
- <p style="color:red;">This is a paragraph.</p>
- </body>
- </html>

Sample Output

This is a heading

This is a paragraph.

- **Fonts**
- The CSS font-family property defines the font to be used for an HTML element:
- **Example**
- <h1 style="font-family:verdana;">This is a heading</h1>

<p style="font-family:courier;">This is a paragraph.</p>

- <!DOCTYPE html>
- <html>
- <body>
- <h1 style="font-family: verdana;">This is a heading</h1>
- <p style="font-family: courier;">This is a paragraph.</p>
- </body>
- </html>

Sample Output

This is a heading

This is a paragraph.

- **Text Size**
- The CSS font-size property defines the text size for an HTML element:
- **Example**
- <h1 style="font-size:300%;">This is a heading</h1>

<p style="font-size:160%;">This is a paragraph.</p>

- **Example**
- <!DOCTYPE html>
- <html>
- <body>
- <h1 style="font-size:300%;">This is a heading</h1>
- <p style="font-size:160%;">This is a paragraph.</p>
- </body>
- </html>

Sample Output

This is a heading

This is a paragraph.

- **Text Alignment**
- The CSS text-align property defines the horizontal text alignment for an HTML element:
- **Example**
- <h1 style="text-align:center;">Centered Heading</h1>
  <p style="text-align:center;">Centered paragraph.</p>
- **Example:**
- <!DOCTYPE html>
- <html>
- <body>
- <h1 style="text-align:center;">Centered Heading</h1>
- <p style="text-align:center;">Centered paragraph.</p>
- </body>

• </html>

Sample Output

Centered Heading

Centered paragraph.

• **Summary**

• Use the **style** attribute for styling HTML elements

• Use **background-color** for background color

• Use **color** for text colors

• Use **font-family** for text fonts

• Use **font-size** for text sizes

• Use **text-align** for text alignment

2.6 HTML Text Formatting

HTML contains several elements for defining text with a special meaning.

Example

This text is bold

This text is italic

This is subscript and superscript

- **Example**
- <!DOCTYPE html>
- <html>
- <body>
- <p>This text is bold</p>
- <p><i>This text is italic</i></p>
- <p>This is_{subscript} and ^{superscript}</p>
- </body>
- </html>
- **HTML Formatting Elements**
- Formatting elements were designed to display special types of text:
- - Bold text
- <strong> - Important text
- <i> - Italic text
- <em> - Emphasized text
- <mark> - Marked text
- <small> - Smaller text
- <del> - Deleted text
- <ins> - Inserted text

• <sub> - Subscript text

• <sup> - Superscript text

•• HTML and <strong> Elements

• The HTML element defines bold text, without any extra importance.

• Example

• This text is bold

• Example

```
<!DOCTYPE html>
<html>
<body>
<p>This text is normal.</p>
<p><b>This text is bold.</b></p>
</body>
</html>
```

Sample Output

This text is normal.

This text is bold.

• The HTML element defines text with strong importance. The content inside is typically displayed in bold.

• Example

• This text is important!

```
<!DOCTYPE html>
<html>
<body>
<p>This text is normal.</p>
<p><strong>This text is important!</strong></p>
</body> </html>
```

Sample Output

This text is normal.

This text is important!

- **HTML <i> and <em> Elements**
- The HTML <i> element defines a part of text in an alternate voice or mood. The content inside is typically displayed in italic.
- **Tip:** The <i> tag is often used to indicate a technical term, a phrase from another language, a thought, a ship name, etc.
- **Example**

```
<i>This text is italic</i>
<!DOCTYPE html>
<html>
<body>
<p>This text is normal.</p>
<p><i>This text is italic.</i></p>
</body>
</html>
```

Sample Output

This text is normal.

This text is italic.

• The HTML element defines emphasized text. The content inside is typically displayed in italic.

• **Tip:** A screen reader will pronounce the words in with an emphasis, using verbal stress.

• **Example**

• This text is emphasized

```
<!DOCTYPE html>
<html>
<body>
<p>This text is normal.</p>
<p><em>This text is emphasized.</em></p>
</body>
</html>
```

Sample Output

This text is normal.

This text is emphasized.

• **HTML <small> Element**

• The HTML <small> element defines smaller text:

• **Example**

• <small>This is some smaller text.</small>

```
<!DOCTYPE html>
<html>
<body>
<p>This is some normal text.</p>
<p><small>This is some smaller text.</small></p>
```

</body>

</html>

Sample Output

This is some normal text.

This is some smaller text.

- **HTML <mark> Element**
- The HTML <mark> element defines text that should be marked or highlighted:
- **Example**
- <p>Do not forget to buy <mark>milk</mark> today.</p>

<!DOCTYPE html>

<html>

<body>

<p>Do not forget to buy <mark>milk</mark> today.</p>

</body>

</html>

Sample Output

Do not forget to buy milk today.

• **HTML <del> Element**

• The HTML <del> element defines text that has been deleted from a document. Browsers will usually strike a line through deleted text:

• **Example**

• <p>My favorite color is ~~blue~~ red.</p>

```
<!DOCTYPE html>
<html>
<body>
<p>My favorite color is <del>blue</del> red.</p>
</body>
</html>
```

Sample Output

My favorite color is ~~blue~~ red.

• **HTML <ins> Element**

• The HTML <ins> element defines a text that has been inserted into a document. Browsers will usually underline inserted text:

• **Example**

• <p>My favorite color
is ~~blue~~ <ins>red</ins>.</p>

```
<!DOCTYPE html>
<html>
<body>
```

```
<p>My favorite color is <del>blue</del>
<ins>red</ins>.</p>
</body>
</html>
```

Sample Output

My favorite color is ~~blue~~ red.

- **HTML <sub> Element**
- The HTML <sub> element defines subscript text. Subscript text appears half a character below the normal line, and is sometimes rendered in a smaller font. Subscript text can be used for chemical formulas, like H2O:

- **Example**
- <p>This is _{subscripted} text.</p>

```
<!DOCTYPE html>
<html>
<body>
<p>This is <sub>subscripted</sub> text.</p>
</body>
</html>
```

Sample Output

This is $_{\text{subscripted}}$ text.

- **HTML <sup> Element**
- The HTML <sup> element defines superscript text. Superscript text appears half a character above the normal line, and is sometimes rendered in a smaller font. Superscript text can be used for footnotes, like WWW[1]:
- **Example**
- <p>This is ^{superscripted} text.</p>

```
<!DOCTYPE html>
<html>
<body>
<p>This is <sup>superscripted</sup> text.</p>
</body>
</html>
```

Sample Output

This is $^{\text{superscripted}}$ text.

HTML Text Formatting Elements

Tag	Description
<b>	Defines bold text
<em>	Defines emphasized text
<i>	Defines a part of text in an alternate voice or mood
<small>	Defines smaller text
<strong>	Defines important text
<sub>	Defines subscripted text
<sup>	Defines superscripted text
<ins>	Defines inserted text

2.7 HTML Comments

HTML comments are not displayed in the browser, but they can help document your HTML source code.

HTML Comment Tags

You can add comments to your HTML source by using the following syntax:

<!-- Write your comments here -->

Notice that there is an exclamation point (!) in the start tag, but not in the end tag.

Note: Comments are not displayed by the browser, but they can help document your HTML source code. With comments you can place notifications and reminders in your HTML code:

- **Example**
- <!-- This is a comment -->

<p>This is a paragraph.</p>

<!-- Remember to add more information here -->

<!DOCTYPE html>

<html>

<body>

```
<!-- This is a comment -->
<p>This is a paragraph.</p>
<!-- Comments are not displayed in the browser -->
</body>
</html>
```

Sample Output

This is a paragraph.

• Comments are also great for debugging HTML, because you can comment out HTML lines of code, one at a time, to search for errors:

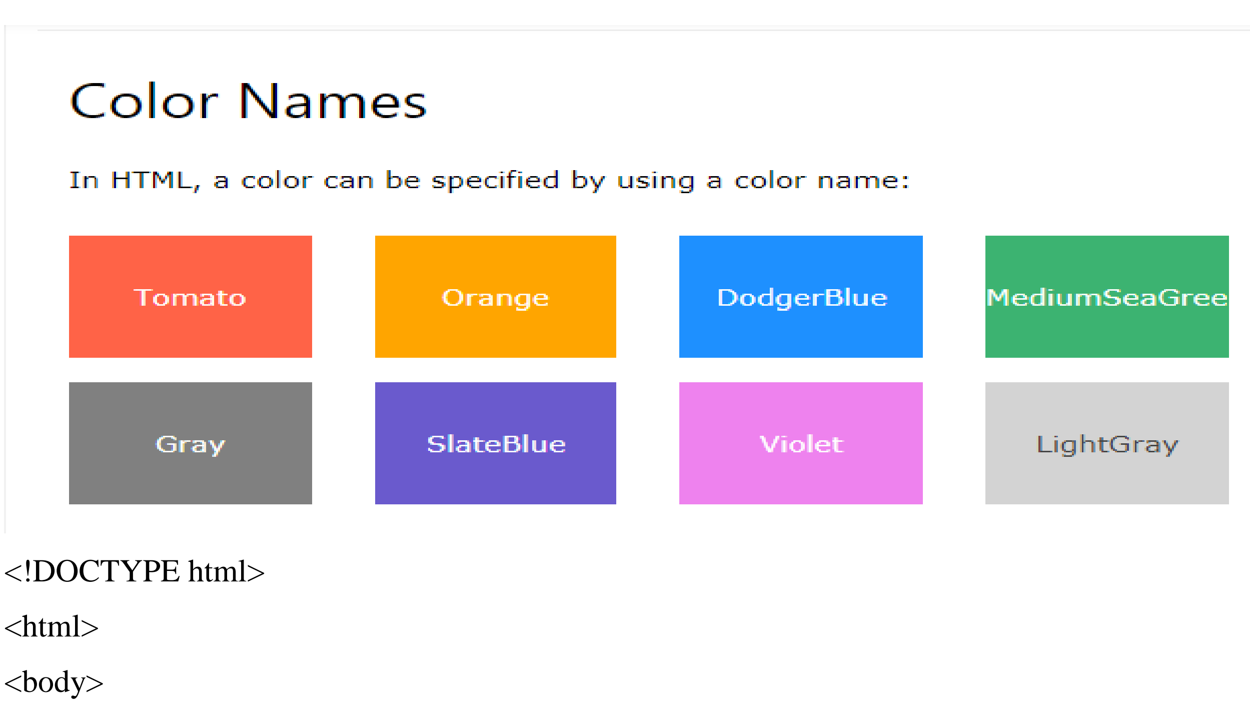
• **Example**

```
• <!-- Do not display this image at the moment
<img border="0" src="pic_trulli.jpg" alt="Trulli">
-->
• <!DOCTYPE html>
• <html>
• <body>
• <!-- Do not display this at the moment
• <img border="0" src="pic_trulli.jpg" alt="Trulli">
• -->
• </body>
• </html>
```

Sample Output

2.8 HTML Colors

HTML colors are specified with predefined color names, or with RGB, HEX, HSL, RGBA, or HSLA values.




```
<!DOCTYPE html>
<html>
<body>
```

```
<h1 style="background-color:Tomato;">Tomato</h1>
<h1 style="background-color:Orange;">Orange</h1>
<h1 style="background
color:DodgerBlue;">DodgerBlue</h1>
<h1 style="background color:MediumSeaGreen;">MediumSeaGreen</h1>
<h1 style="background-color:Gray;">Gray</h1>
<h1 style="background-color:SlateBlue;">SlateBlue</h1>
<h1 style="background-color:Violet;">Violet</h1>
<h1 style="background-color:LightGray;">LightGray</h1>
</body>
</html>
```

Sample Output:

Tomato

Orange

DodgerBlue

MediumSeaGreen

Gray

SlateBlue

- **Background Color**

```
<!DOCTYPE html>
<html>
<body>
<h1 style="background-color:DodgerBlue;">Hello World</h1>
<p style="background-color:Tomato;">
```

- Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

• Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

</p>

</body>

</html>

Sample Output

Hello World

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

• **Text Color**

<!DOCTYPE html>

<html>

<body>

<h3 style="color:Tomato;">Hello World

<p style="color:DodgerBlue;">Lorem ipsum dolor sit amet,consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.</p>

<p style="color:MediumSeaGreen;">Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.</p>

</body>

```
</html>
```

Sample Output

Hello World

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

- **Border Color**
- You can set the color of borders

```
<!DOCTYPE html>
<html>
<body>
<h1 style="border: 2px solid Tomato;">Hello World</h1>
<h1 style="border: 2px solid DodgerBlue;">Hello World</h1>
<h1 style="border: 2px solid Violet;">Hello World</h1>
</body>
</html>
```

Sample Output

Hello World

Hello World

Hello World

Color Values

In HTML, colors can also be specified using RGB values, HEX values, HSL values, RGBA values, and HSLA values.

The following three <div> elements have their background color set with RGB, HEX, and HSL values:

rgb(255, 99, 71)

#ff6347

hsl(9, 100%, 64%)

The following two <div> elements have their background color set with RGBA and HSLA values, which adds an Alpha channel to the color (here we have 50% transparancy):

rgba(255, 99, 71, 0.5)

hsla(9, 100%, 64%, 0.5)

```
<!DOCTYPE html>
<html>
<body>
<p>Same as color name "Tomato":</p>
<h1 style="background-color:rgb(255, 99, 71);">rgb(255, 99, 71)</h1>
<h1 style="background-color:#ff6347;">#ff6347</h1>
<h1 style="background-color:hsl(9, 100%, 64%);">hsl(9, 100%,
64%)</h1>
<p>Same as color name "Tomato", but 50% transparent:</p>
<h1 style="background-color:rgba(255, 99, 71, 0.5);">rgba(255, 99, 71, 0.5)</h1>
<h1 style="background-color:hsla(9, 100%, 64%, 0.5);">hsla(9, 100%, 64%, 0.5)</h1>
<p>In addition to the predefined color names, colors can be specified using RGB, HEX, HSL, or
even transparent colors using RGBA or HSLA color values.</p>
</body>
</html>
```

Sample Output

Same as color name "Tomato":

rgb(255, 99, 71)

#ff6347

hsl(9, 100%, 64%)

Same as color name "Tomato", but 50% transparent:

rgba(255, 99, 71, 0.5)

hsla(9, 100%, 64%, 0.5)

In addition to the predefined color names, colors can be specified using RGB, HEX, HSL, or even transparent colors using RGBA or HSLA color values.

2.9 HTML Links

HTML Links - Hyperlinks

• HTML links are hyperlinks.

• You can click on a link and jump to another document.

• When you move the mouse over a link, the mouse arrow will turn into a little hand.

• **Note:** A link does not have to be text. A link can

be an image or any other HTML element!

HTML Links - Syntax

The HTML <a> tag defines a hyperlink. It has the following syntax:

link text• The most important attribute of the <a> element is the href attribute, which indicates the link's destination.

• The *link text* is the part that will be visible to the reader.

• Clicking on the link text, will send the reader to the specified URL address.

• **Example**

• This example shows how to create a link to W3Schools.com:

```
<!DOCTYPE html>
<html>
<body>
<h1>HTML Links</h1>
<p><a href="https://www.w3schools.com/">Visit
W3Schools.com!</a></p>
</body>
</html>
```

Sample Output:

HTML Links

Visit W3Schools.com!

• By default, links will appear as follows in all browsers:

• An unvisited link is underlined and blue

• A visited link is underlined and purple

• An active link is underlined and red

• **HTML Links - The target Attribute**

• By default, the linked page will be displayed in the current browser window. To change this, you must specify another target for the link.

• The target attribute specifies where to open the linked document.

• The target attribute can have one of the following values:

• _self - Default. Opens the document in the same window/tab as it was clicked

• _blank - Opens the document in a new window or tab

• _parent - Opens the document in the parent frame

• _top - Opens the document in the full body of the Window

• **Example**

```
<!DOCTYPE html>
<html>
<body>
<h2>The target Attribute</h2>
<a href="https://www.w3schools.com/"
target="_blank">Visit W3Schools!</a>
<p>If target="_blank", the link will open in a new
browser window or tab.</p>
</body>
</html>
```

Sample Output

The target Attribute

Visit W3Schools!

If target="_blank", the link will open in a new browser window or tab.

• **Absolute URLs vs. Relative URLs**

• Both examples above are using an **absolute URL** (a full web address) in the href attribute.

• A local link (a link to a page within the same website) is specified with a **relative URL** (without the "https://www" part):

• **Example**

```
<!DOCTYPE html>
<html>
<body>
<h2>Absolute URLs</h2>
<p><a href="https://www.w3.org/">W3C</a></p>
<p><a href="https://www.google.com/">Google</a></p>
<h2>Relative URLs</h2>
<p><a href="html_images.asp">HTML Images</a></p>
<p><a href="/css/default.asp">CSS Tutorial</a></p>
</body>
</html>
```

Sample Output

Absolute URLs

W3C

Google

Relative URLs

HTML Images

CSS Tutorial

• **HTML Links - Use an Image as a Link**

• To use an image as a link, just put the <img> tag inside the <a> tag:

• **Example**

```
<!DOCTYPE html>
<html>
<body>
<h2>Image as a Link</h2>
<p>The image below is a link. Try to click on it.</p>
<a href="default.asp"><img src="smiley.gif" alt="HTML tutorial"
style="width:42px;height:42px;"></a>
</body>
</html>
```

Sample Output

Image as a Link

The image below is a link. Try to click on it.

• **Link to an Email Address**

• Use mailto: inside the href attribute to create a link that opens the user's email program (to let them send a new email):

Example:

```
<!DOCTYPE html>
```

```
<html>
<body>
<h2>Link to an Email Address</h2>
<p>To create a link that opens in the user's email program (to let them send a new email), use
mailto: inside the href attribute:</p>
<p><a href="mailto:someone@example.com">Send email</a></p>
</body>
</html>
```

Sample Output

Link to an Email Address

To create a link that opens in the user's email program (to let them send a new email), use mailto: inside the href attribute:

Send email

- **Button as a Link**
- To use an HTML button as a link, you have to add some JavaScript code.
- JavaScript allows you to specify what happens at certain events, such as a click of a button:
- **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>Button as a Links</h2>
<p>Click the button to go to the HTML tutorial.</p>
<button onclick="document.location='default.asp'">HTML Tutorial</button>
</body>
</html>
```

Sample Output

Button as a Links

Click the button to go to the HTML tutorial.

HTML Tutorial

- **Link Titles**
- The title attribute specifies extra information about an element. The information is most often shown as a tooltip text when the mouse moves over the element.
- **Example**
- <!DOCTYPE html>
- <html lang="en-US">
- <body>
- <h2>Link Titles</h2>
- <p>The title attribute specifies extra information about an element. The information is most often shown as a tooltip text when the mouse moves over the element.</p>
- Visit our HTML Tutorial
- </body>
- </html>

Sample Output

Link Titles

The title attribute specifies extra information about an element. The information is most often shown as a tooltip text when the mouse moves over the element.

Visit our HTML Tutorial

- **More on Absolute URLs and Relative URLs**
- **Example: Use a full URL to link to a web page:**

```
<!DOCTYPE html>
<html>
<body>
<h2>External Paths</h2>
<p>This example uses a full URL to link to a web page:</p>
<p><a
href="https://www.w3schools.com/html/default. asp">HTML tutorial</a></p>
</body>
</html>
```

Sample Output

External Paths

This example uses a full URL to link to a web page:

HTML tutorial

• **Example**

• Link to a page located in the html folder on the current web site:

```
<!DOCTYPE html>
<html>
<body>
<h2>External Paths</h2>
<p>This example links to a page located in the html folder on the current web site:</p>
<p><a href="/html/default.asp">HTML tutorial</a></p>
</body>
</html>
```

Sample Output

External Paths

This example links to a page located in the html folder on the current web site:

HTML tutorial

- **Example**
- Link to a page located in the same folder as the current page:
- HTML tutorial
- <!DOCTYPE html>
- <html>
- <body>
- <h2>External Paths</h2>
- <p>This example links to a page located in the same folder as the current page:</p>
- <p>HTML tutorial</p>
- </body>
- </html>

Sample Output

External Paths

This example links to a page located in the same folder as the current page:

HTML tutorial

- **HTML Links - Different Colors**
- An HTML link is displayed in a different color depending on whether it has been visited, is unvisited, or is active.
- **HTML Link Colors**
- By default, a link will appear like this (in all browsers):
- An unvisited link is underlined and blue
- A visited link is underlined and purple
- An active link is underlined and red
- You can change the link state colors, by using CSS:
- **Example**
- Here, an unvisited link will be green with no underline. A visited link will be pink with no underline. An active link will be yellow and underlined. In addition, when mousing over a link (a:hover) it will become red and underlined:

```
<!DOCTYPE html>
<html>
<head>
<style>
a:link {
color: green;
background-color: transparent;
text-decoration: none;
}a:visited
{
```

```
color: pink; background-color: transparent; text-decoration: none;
}
a:hover {
color: red; background-color: transparent; text-decoration: underline;
}
a:active {
color: yellow; background-color: transparent; text-decoration:
underline;
} </style> </head> <body>
<h2>Link Colors</h2>
<p>You can change the default colors of links</p>
<a href="html_images.asp" target="_blank">HTML Images</a>
</body> </html>
```

Sample Output

Link Colors

You can change the default colors of links

HTML Images

- **Link Buttons**
- A link can also be styled as a button, by using CSS:
- Example

```
<!DOCTYPE html>
```

```
<html>
<head>
<style>
a:link, a:visited {
background-color: #f44336;
color: white;
padding: 15px 25px;
text-align: center;
text-decoration: none;
display: inline-block;
}a:hover, a:active {
background-color: red;
}
</style>
</head>
<body>
<h2>Link Button</h2>
<p>A link styled as a button:</p>
<a href="default.asp" target="_blank">This is a link</a>
</body>
</html>
```

Sample Output

Link Button

A link styled as a button:

This is a link

• **HTML Links - Create Bookmarks**

• HTML links can be used to create bookmarks, so that readers can jump to specific parts of a web page.

• **Create a Bookmark in HTML**

• Bookmarks can be useful if a web page is very long.

• To create a bookmark - first create the bookmark, then add a link to it.

• When the link is clicked, the page will scroll down or up to the location with the bookmark.

• **Example**

• First, use the id attribute to create a bookmark:

• <h2 id="C4">Chapter 4</h2>

• Then, add a link to the bookmark ("Jump to Chapter 4"), from within the same page:

• **Example**

• Jump to Chapter 4

• You can also add a link to a bookmark on another page:

• Jump to Chapter 4

2.10 HTML Images

Images can improve the design and the appearance of a web page.

Example

```
<!DOCTYPE html>
<html>
<body>
<h2>HTML Image</h2>
<img src="pic_trulli.jpg" alt="Trulli" width="500"
height="333">
</body>
</html>
```

Sample Output

- **Example**
- <!DOCTYPE html>
- <html>
- <body>
- <h2>HTML Image</h2>
- <img src="img_girl.jpg" alt="Girl in a jacket"
width="500" height="600">
- </body>
- </html>

Sample Output

• **HTML Images Syntax**

• The HTML <img> tag is used to embed an image in a web page.

• Images are not technically inserted into a web page; images are linked to web pages. The <img> tag creates a holding space for the referenced image.

• The <img> tag is empty, it contains attributes only, and does not have a closing tag.

• The <img> tag has two required attributes:

• src - Specifies the path to the image

• alt - Specifies an alternate text for the image

• **Syntax**

• <img src="*url*" alt="*alternatetext*">• **The src Attribute**

• The required src attribute specifies the path (URL) to the image.

• **Note:** When a web page loads; it is the browser, at that moment, that gets the image from a web server and inserts it into the page. Therefore, make sure that the image actually stay in the same spot in relation to the web page, otherwise your visitors will get a broken link icon. The broken link icon and the alt text are shown if the browser cannot find the image.

• **Example**

• <!DOCTYPE html>

• <html>

• <body>

• <h2>Alternative text</h2>

• <p>The alt attribute should reflect the image content, so users who cannot see the image gets an understanding of what the image contains:</p>

• <img src="img_chania.jpg" alt="Flowers in Chania" width="460" height="345">

• </body>

• </html>

Sample Output

• **The alt Attribute**

• The required alt attribute provides an alternate text for an image, if the user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader).

• The value of the alt attribute should describe the image:

• Example

• <!DOCTYPE html>

• <html>

• <body>

• <h2>Alternative text</h2>

• <p>The alt attribute should reflect the image content, so users who cannot see the image gets an understanding of what the image contains:</p>

• <img src="img_chania.jpg" alt="Flowers in Chania"
width="460" height="345">

• </body>

• </html>

Sample Output

Alternative text

The alt attribute should reflect the image content, so users who cannot see the image gets an understanding of what the image contains:

• Image Size - Width and Height

• You can use the style attribute to specify the width and height of an image

• **Example**

• <!DOCTYPE html>

• <html>

• <body>

• <h2>Image Size</h2>

• <p>Here we use the style attribute to specify the width and height of an image:</p>

• <img src="img_girl.jpg" alt="Girl in a jacket" style="width:500px;height:600px;">

• </body>

• </html>

Sample Output

Image Size

Here we use the style attribute to specify the width and height of an image:

• Alternatively, you can use the width and height attributes:

• <!DOCTYPE html>

• <html>

• <body>

• <h2>Image Size</h2>

• <p>Here we specify the width and height of an image with the width and height attributes:</p>

• <img src="img_girl.jpg" alt="Girl in a jacket" width="500" height="600">

• </body>

• </html>

Sample Output

Image Size

Here we specify the width and height of an image with the width and height attributes:

• **Width and Height, or Style?**

• The width, height, and style attributes are all valid in HTML.

• However, we suggest using the style attribute.

It prevents styles sheets from changing the size of images:

• **Example**

• <!DOCTYPE html> <html> <head>

• <style> /* This style sets the width of all images to 100%: */

• img {

•

width: 100%;

• }

• </style> </head> <body>

• <h2>Width/Height Attributes or Style?</h2>

• <p>The first image uses the width attribute (set to 128 pixels), but the style in the head section overrides it, and sets the width to 100%.</p>

• <img src="html5.gif" alt="HTML5 Icon" width="128" height="128">

• <p>The second image uses the style attribute to set the width to 128 pixels, this will not be overridden by the style in the head section:</p>

• <img src="html5.gif" alt="HTML5 Icon" style="width:128px;height:128px;">

• </body> </html>

Sample Output

Width/Height Attributes or Style?

The first image uses the width attribute (set to 128 pixels), but the style in the head section overrides it, and sets the width to 100%.

The second image uses the style attribute to set the width to 128 pixels, this will not be overridden by the style in the head section:

Common Image Formats

Here are the most common image file types, which are supported in all browsers (Chrome, Edge, Firefox, Safari, Opera):

Abbreviation	File Format	File Extension
APNG	Animated Portable Network Graphics	.apng
GIF	Graphics Interchange Format	.gif
ICO	Microsoft Icon	.ico, .cur
JPEG	Joint Photographic Expert Group image	.jpg, .jpeg, .jfif, .pjpeg, .pjp
PNG	Portable Network Graphics	.png
SVG	Scalable Vector Graphics	.svg

2.11 HTML Lists

HTML lists allow web developers to group a set of related items in lists.

An unordered HTML list:

Item

Item

Item

Item

An ordered HTML list:

First item

Second item

Third item

Fourth item• **Unordered HTML List**

• An unordered list starts with the <ul> tag. Each list item starts with the <li> tag.

• The list items will be marked with bullets (small black circles) by default:

• **Example**

• <!DOCTYPE html>

• <html>

• <body>

• <h2>An unordered HTML list</h2>

•

• Coffee

• Tea

• Milk

•

• </body>

• </html>

Sample Output

An unordered HTML list

- Coffee
- Tea
- Milk

- **Ordered HTML List**
- An ordered list starts with the <ol> tag. Each list item starts with the <li> tag.
- The list items will be marked with numbers by default:
- **Example:**
- <!DOCTYPE html>
- <html>
- <body>
- <h2>An ordered HTML list</h2>
-
- Coffee
- Tea
- Milk
-
- </body>
- </html>

Sample Output

An ordered HTML list

1. Coffee
2. Tea
3. Milk

• **HTML Description Lists**

• HTML also supports description lists.

• A description list is a list of terms, with a description of each term.

• The <dl> tag defines the description list, the <dt> tag defines the term (name), and the <dd> tag describes each term:

• **Example• Example:**

• <!DOCTYPE html>

• <html>

• <body>

• <h2>A Description List</h2>

• <dl>

• <dt>Coffee</dt>

• <dd>- black hot drink</dd>

• <dt>Milk</dt>

• <dd>- white cold drink</dd>

- </dl>
- </body>
- </html>

Sample Output

> **A Description List**
>
> Coffee
>   - black hot drink
> Milk
>   - white cold drink

HTML List Tags

Tag	Description
	Defines an unordered list
	Defines an ordered list
	Defines a list item
<dl>	Defines a description list
<dt>	Defines a term in a description list
<dd>	Describes the term in a description list

2.12 HTML Forms

An HTML form is used to collect user input. The user input is most often sent to a server for processing.

Example:

```
<!DOCTYPE html>
<html> <body> <h2>HTML Forms</h2>
<form action="/action_page.php">
<label for="fname">First name:</label><br>
<input type="text" id="fname" name="fname" value="John"><br>
<label for="lname">Last name:</label><br>
<input type="text" id="lname" name="lname" value="Doe"><br><br>
<input type="submit" value="Submit">
</form>
<p>If you click the "Submit" button, the form-data will be sent to a page called
"/action_page.php".</p>
</body> </html>
```

Sample Output

HTML Forms

First name:
John
Last name:
Doe

Submit

If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".

• **The <form> Element**

• The HTML <form> element is used to create an HTML form for user input:

<form>

.

form elements

.

</form>

• The <form> element is a container for different types of input elements, such as: text fields, checkboxes, radio buttons, submit buttons, etc.

• All the different form elements are covered in the next chapter:

HTML Form Elements.

The <input> Element

The HTML <input> element is the most used form element. An <input> element can be displayed in many ways, depending on the type attribute.

Here are some examples:

Type	Description
<input type="text">	Displays a single-line text input field
<input type="radio">	Displays a radio button (for selecting one of many choices)
<input type="checkbox">	Displays a checkbox (for selecting zero or more of many choices)
<input type="submit">	Displays a submit button (for submitting the form)
<input type="button">	Displays a clickable button

• **Text Fields**

• The <input type="text"> defines a single-line input field for text input**.**

• **Example**

• <!DOCTYPE html>

- <html> <body>
- <h2>Text input fields</h2>
- <form>
- <label for="fname">First name:</label>
- <input type="text" id="fname" name="fname" value="John">
- <label for="lname">Last name:</label>
- <input type="text" id="lname" name="lname" value="Doe">
- </form>
- <p>Note that the form itself is not visible.</p>
- <p>Also note that the default width of text input fields is 20 characters.</p>
- </body> </html>

Sample Output

Text input fields

First name:

John

Last name:

Doe

Note that the form itself is not visible.

Also note that the default width of text input fields is 20 characters.

This is how the HTML code above will be displayed in a browser:

First name:

Last name:

Note: The form itself is not visible. Also note that the default width of an input field is 20 characters.

• **The <label> Element**

• Notice the use of the <label> element in the example above.

• The <label> tag defines a label for many form elements.

• The <label> element is useful for screen-reader users, because the screen-reader will read out loud the label when the user focus on the input element.

• The <label> element also help users who have difficulty clicking on very small regions (such as radio buttons or checkboxes) - because when the user clicks the text within the <label> element, it toggles the radio button/checkbox.

• The for attribute of the <label> tag should be equal to the id attribute of the <input> element to bind them together.

• **Radio Buttons**

• The <input type="radio"> defines a radio button.

• Radio buttons let a user select ONE of a limited number of choices.

- **Example:**
- <!DOCTYPE html>
- <html>
- <body>
- <h2>Radio Buttons</h2>
- <form>
- <input type="radio" id="male" name="gender" value="male">
- <label for="male">Male</label>
- <input type="radio" id="female" name="gender" value="female">
- <label for="female">Female</label>
- <input type="radio" id="other" name="gender" value="other">
- <label for="other">Other</label>
- </form>
- </body>
- </html>

Sample Output

Radio Buttons

○ Male
○ Female
○ Other

- **Checkboxes**
- The <input type="checkbox"> defines a **checkbox**.
- Checkboxes let a user select ZERO or MORE options of a limited number of choices.

- **Example**
- <!DOCTYPE html>
- <html> <body>
- <h2>Checkboxes</h2>
- <p>The input type="checkbox" defines a checkbox:</p>
- <form action="/action_page.php">
- <input type="checkbox" id="vehicle1" name="vehicle1" value="Bike">
- <label for="vehicle1"> I have a bike</label>
- <input type="checkbox" id="vehicle2" name="vehicle2" value="Car">
- <label for="vehicle2"> I have a car</label>
- <input type="checkbox" id="vehicle3" name="vehicle3" value="Boat">
- <label for="vehicle3"> I have a boat</label>

- <input type="submit" value="Submit">
- </form> </body> </html>

Sample Output

Checkboxes

The **input type="checkbox"** defines a checkbox:

☐ I have a bike
☐ I have a car
☐ I have a boat

Submit

- **The Submit Button**
- The <input type="submit"> defines a button for submitting the form data to a form handler.
- The form-handler is typically a file on the server with a script for processing input data.
- The form-handler is specified in the form's action attribute.

- **Example:**
- <!DOCTYPE html>
- <html>
- <body>
- <h2>HTML Forms</h2>
- <form action="/action_page.php">
- <label for="fname">First name:</label>
- <input type="text" id="fname" name="fname" value="John">
- <label for="lname">Last name:</label>
- <input type="text" id="lname" name="lname" value="Doe">

- <input type="submit" value="Submit">
- </form>
- <p>If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".</p>
- </body>
- </html>

Sample Output

HTML Forms

First name:

John

Last name:

Doe

Submit

If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".

• **The Action Attribute**

• The action attribute defines the action to be performed when the form is submitted.

• Usually, the form data is sent to a file on the server when the user clicks on the submit button.

• In the example above, the form data is sent to a file called "action_page.php". This file contains a server-side script that handles the form data:

• <form **action="/action_page.php"**>

• If the action attribute is omitted, the action is set to the current page.

The Target Attribute

The target attribute specifies where to display the response that is received after submitting the form.

The target attribute can have one of the following values:

Value	Description
_blank	The response is displayed in a new window or tab
_self	The response is displayed in the same frame
_parent	The response is displayed in the parent frame
_top	The response is displayed in the full body of the window
framename	The response is displayed in a named iframe

The default value is `_self` which means that the response will open i the current window.

- The default value is _self which means that the response will open i the current window.
- **Example:**
- <!DOCTYPE html>
- <html> <body>
- <h2>The form target attribute</h2>
- <p>When submitting this form, the result will be opened in a new browser tab:</p>
- <form action="/action_page.php" target="_blank">
- <label for="fname">First name:</label>
- <input type="text" id="fname" name="fname" value="John">
- <label for="lname">Last name:</label>
- <input type="text" id="lname" name="lname" value="Doe">

- <input type="submit" value="Submit">
- </form> </body> </html>

Sample Output

The form target attribute

When submitting this form, the result will be opened in a new browser tab:

First name:
John
Last name:
Doe

Submit

- **The Method Attribute**
- The method attribute specifies the HTTP method to use used when submitting the form data.
- The form-data can be sent as URL variables (with method="get") or as HTTP post transaction (with method="post").
- The default HTTP method when submitting form data is GET.

- **Example: 1**
- <!DOCTYPE html>
- <html> <body>
- <h2>The method Attribute</h2>
- <p>This form will be submitted using the GET method:</p>
- <form action="/action_page.php" target="_blank" method="get">
- <label for="fname">First name:</label>
- <input type="text" id="fname" name="fname" value="John">

- <label for="lname">Last name:</label>
- <input type="text" id="lname" name="lname" value="Doe">

- <input type="submit" value="Submit">
- </form>
- <p>After you submit, notice that the form values is visible in the address bar of the new browser tab.</p>
- </body> </html>

Sample Output

The method Attribute

This form will be submitted using the GET method:

First name:
John
Last name:
Doe

Submit

After you submit, notice that the form values is visible in the address bar of the new browser tab.

- **Example : 2**
- <!DOCTYPE html>
- <html> <body>
- <h2>The method Attribute</h2>
- <p>This form will be submitted using the POST method:</p>
- <form action="/action_page.php" target="_blank" method="post">
- <label for="fname">First name:</label>
- <input type="text" id="fname" name="fname" value="John">
- <label for="lname">Last name:</label>

• <input type="text" id="lname" name="lname"
value="Doe">

• <input type="submit" value="Submit">

• </form>

• <p>After you submit, notice that, unlike the GET method, the form values is NOT visible in the address bar of the new browser tab.</p>

• </body> </html>

Sample Output

The method Attribute

This form will be submitted using the POST method:

First name:
John
Last name:
Doe

Submit

After you submit, notice that, unlike the GET method, the form values is NOT visible in the address bar of the new browser tab.

• **Notes on GET:**

• Appends the form data to the URL, in name/value pairs • NEVER use GET to send sensitive data! (the submitted form data is visible in the URL!)

• The length of a URL is limited (2048 characters)

• Useful for form submissions where a user wants to bookmark the result

• GET is good for non-secure data, like query strings in Google

• **Notes on POST:**

• Appends the form data inside the body of the HTTP request (the submitted form data is not shown in the URL)

• POST has no size limitations, and can be used to send large amounts of data.

• Form submissions with POST cannot be bookmarked

• **Tip:** Always use POST if the form data contains sensitive or personal information!

• **The Name Attribute**

• Each input field must have a name attribute to be submitted.

• If the name attribute is omitted, the value of that input field will not be sent at all.

• **Example:**

• <!DOCTYPE html>

• <html> <body>

• <h2>The name Attribute</h2>

• <form action="/action_page.php">

• <label for="fname">First name:</label>

• <input type="text" id="fname" value="John">

• <input type="submit" value="Submit">

• </form>

• <p>If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".</p>

• <p>Notice that the value of the "First name" field will not be submitted, because the input element does not have a name attribute.</p>

• </body> </html>

Sample Output

The name Attribute

First name:

John

Submit

If you click the "Submit" button, the form-data will be sent to a page called "/action_page.php".

Notice that the value of the "First name" field will not be submitted, because the input element does not have a name attribute.

2.13 HTML <frame> tag

HTML <frame> tag define the particular area within an HTML file where another HTML web page can be displayed. A <frame> tag is used with <frameset>, and it divides a webpage into multiple sections or frames, and each frame can contain different web pages.

Note: Do not use HTML <frame> tag as it is not supported in HTML5, instead you can use <iframe> or <div> with CSS to achieve similar effects in HTML.

Syntax

< frame src = "URL" **>**• Example 1

• Create Vertical frames:

• <!DOCTYPE html>

• **<html>**

• **<head>**

- **<title>**Frame tag**</title>**
- **</head>**
- **<frameset** cols="25%,50%,25%">
- **<frame** src="frame1.html" >
- **<frame** src="frame2.html">
- **<frame** src="frame3.html">
- **</frameset>**
- **</html>**

Output:

Frame1.html

```
<!DOCTYPE html>
<html>
<head>
   <style>
     div{
                          background-color: #7fffd4;
       height: 500px;
      }
   </style>
</head>
<body>
   <div>
      <h2>This is first frame</h2>
   </div>
 </body>
</html>
```

Frame2.html

```
<!DOCTYPE html>
<html>
<head>
   <style>
     div{
       background-color: #2f4f4f;
       height: 500px;

     }
   </style>
</head>
<body>
   <div>
      <h2>This is Second frame</h2>
   </div>
 </body>
</html>
```

Frame3.html

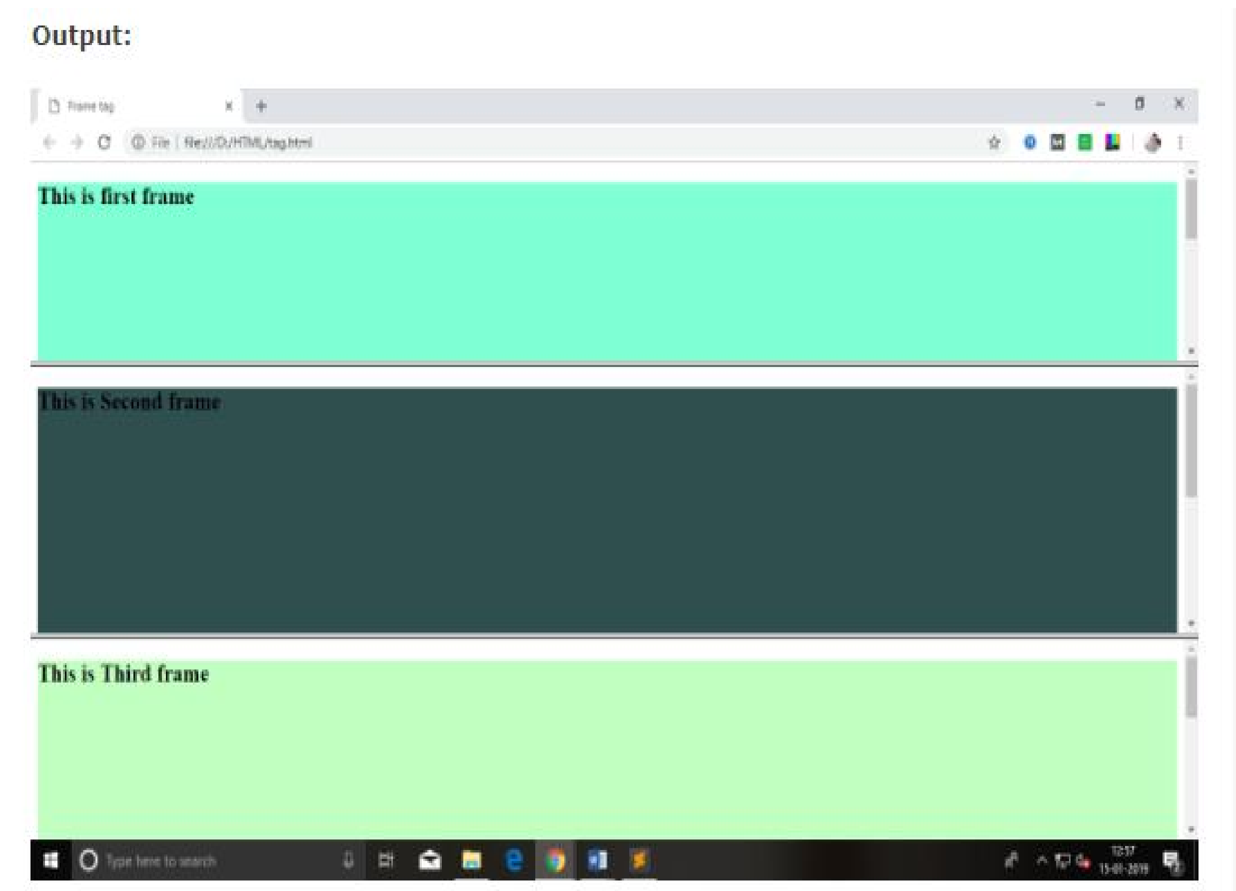
```
<!DOCTYPE html>
<html>
<head>
   <style>
      div{
        background-color: #c1ffc1;
        height: 500px;
                     }
   </style>
</head>
<body>
              <div>
         <h2>This is Third frame</h2>
   </div>
 </body>
</html>
```

Example 2:

Create Horizontal frames:

```
<!DOCTYPE html>
<html>
<head>
   <title>Frame tag</title>
</head>
  <frameset rows="30%, 40%, 30%">
   <frame name="top" src="frame1.html" >
   <frame name="main" src="frame2.html">
   <frame name="bottom" src="frame3.html">
  </frameset>
</html>
```

Output:



Attribute	Value	Description
frameborder	0 1	It specifies whether to display a border around the frame or not, and its default value is 1
longdsec	URL	It specifies a page which contains the long description of the content of the frame.
marginheight	pixels	It specifies the top and bottom margins of the frame.
marginwidth	pixels	It defines the height of the margin between frames.
name	text	It is used to assign the name to the frame.
noresize	noresize	It is used to prevent resizing of the frame by the user.
scrolling	yes no auto	It specifies the existence of the scrollbar for overflowing content.
src	URL	It specifies the URL of the document which we want to display in a frame.

Supporting Browsers

Element	Chrome	IE	Firefox	Opera	Safari
<frame>	Yes	Yes	Yes	Yes	Yes

2.14 HTML Tables

HTML tables allow web developers to arrange data into rows and columns.

Define an HTML Table

The <table> tag defines an HTML table. Each table row is defined with a <tr> tag. Each table header is defined with a <th> tag. Each table data/cell is defined with a <td> tag.

By default, the text in <th> elements are bold and centered.

By default, the text in <td> elements are regular and left aligned.

Note: The <td> elements are the data containers of the table. They can contain all sorts of HTML elements; text, images, lists, other tables, etc.

```
<!DOCTYPE html>
<html> <body>
<h2>Basic HTML Table</h2>
<table style="width:100%">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr>
<tr> <td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td>
<td>Jackson</td> <td>94</td>
</tr>
<tr> <td>John</td>
<td> Doe</td> <td>80</td>
</tr> </table>
</body>
</html>
```

Sample Output

Basic HTML Table

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

- **HTML Table - Add a Border**
- To add a border to a table, use the CSS border property:

```
<!DOCTYPE html> <html>
<head> <style>
table, th, td {
border: 1px solid black;
}
</style> </head> <body>
<h2>Table With Border</h2>
<p>Use the CSS border property to add a border to the table.</p>
<table style="width:100%"> <tr>
<th>Firstname</th> <th>Lastname</th> <th>Age</th>
</tr>
<tr> <td>Jill</td> <td>Smith</td> <td>50</td> </tr>
<tr>
<td>Eve</td> <td>Jackson</td> <td>94</td> </tr>
<tr> <td>John</td> <td>Doe</td> <td>80</td>
```

```
</tr> </table> </body> </html>
```

Sample Output

Table With Border

Use the CSS border property to add a border to the table.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

• **HTML Table - Collapsed Borders**

• To let the borders collapse into one border, add the CSS border-collapse property:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
</style> </head>
<body>
<h2>Collapsed Borders</h2>
<p>If you want the borders to collapse into one border, add the CSS border-collapse
property.</p><table style="width:100%">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
```

```
</tr> <tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td> <td>Jackson</td>
<td>94</td> </tr>
<tr> <td>John</td> <td>Doe</td>
<td>80</td>
</tr> </table>
</body>
</html>
```

Sample Output

Collapsed Borders

If you want the borders to collapse into one border, add the CSS border-collapse property.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

- **HTML Table - Add Cell Padding**
- Cell padding specifies the space between the cell content and its borders.
- If you do not specify a padding, the table cells will be displayed without padding.
- To set the padding, use the CSS padding property:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
th, td {
padding: 15px;
}
</style> </head>
<body>
<h2>Cellpadding</h2>
<p>Cell padding specifies the space between the cell content and its borders.</p><table
style="width:100%">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr> <tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td>
<td>Jackson</td> <td>94</td>
</tr> <tr> <td>John</td>
<td>Doe</td> <td>80</td>
</tr> </table>
<p><strong>Tip:</strong> Try to change the padding to
5px.</p>
</body>
</html>
```

Sample Output CellPadding

Cell padding specifies the space between the cell content and its borders.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

Tip: Try to change the padding to 5px.

- **HTML Table - Left-align Headings**
- By default, table headings are bold and centered.
- To left-align the table headings, use the CSS text-align property:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black; border-collapse: collapse;
}
th, td {
padding: 5px;
}
th {
text-align: left;
}
</style> </head> <body>
<h2>Left-align Headings</h2>
<p>To left-align the table headings, use the CSS text-align
property.</p><table style="width:100%">
```

```
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr> <tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td> <td>Jackson</td>
<td>94</td> </tr>
<tr> <td>John</td> <td>Doe</td>
<td>80</td> </tr>
</table>
</body> </html>
```

Sample Output

Left-align Headings

To left-align the table headings, use the CSS text-align property.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

- **HTML Table - Add Border Spacing**
- Border spacing specifies the space between the cells.
- To set the border spacing for a table, use the CSS border spacing property:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
padding: 5px;
```

```
}
table {
border-spacing: 15px;
}
</style> </head> <body>
<h2>Border Spacing</h2>
<p>Border spacing specifies the space between the cells.</p><table style="width:100%">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr> <tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td> <td>Jackson</td>
<td>94</td> </tr>
<tr> <td>John</td> <td>Doe</td>
<td>80</td> </tr> </table>
<p><strong>Tip:</strong> Try to change the border
spacing to 5px.</p>
</body>
</html>
```

Sample Output Border Spacing

Border spacing specifies the space between the cells.

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

Tip: Try to change the border-spacing to 5px.

- **HTML Table - Cell that Span Many Columns**
- To make a cell span more than one column, use the colspan attribute:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
th, td {
padding: 5px;
text-align: left;
}
</style>
</head><body>
<h2>Cell that spans two columns</h2>
<p>To make a cell span more than one column, use the colspan attribute.</p>
<table style="width:100%">
<tr> <th>Name</th>
<th colspan="2">Telephone</th>
</tr> <tr>
<td>Bill Gates</td>
<td>55577854</td>
<td>55577855</td>
</tr>
</table>
</body>
</html>
```

Sample Output

Cell that spans two columns

To make a cell span more than one column, use the colspan attribute.

Name	Telephone	
Bill Gates	55577854	55577855

• **HTML Table - Cell that Span Many Rows**

• To make a cell span more than one row, use the rowspan attribute:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
th, td {
padding: 5px;
text-align: left;
}
</style> </head><body>
<h2>Cell that spans two rows</h2>
<p>To make a cell span more than one row, use the rowspan attribute.</p>
<table style="width:100%">
<tr> <th>Name:</th>
```

```
<td>Bill Gates</td>
</tr> <tr>
<th rowspan="2">Telephone:</th>
<td>55577854</td> </tr>
<tr> <td>55577855</td>
</tr> </table>
</body>
</html>
```

Sample Output

Cell that spans two rows

To make a cell span more than one row, use the rowspan attribute.

Name:	Bill Gates
Telephone:	55577854
	55577855

• **HTML Table - Add a Caption**

• To add a caption to a table, use the <caption> tag:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
```

```
th, td {
padding: 5px; text-align: left;
}
</style> </head>
<body>
<h2>Table Caption</h2>
<p>To add a caption to a table, use the caption tag.</p><table style="width:100%">
<caption>Monthly savings</caption>
<tr> <th>Month</th>
<th>Savings</th> </tr>
<tr> <td>January</td>
<td>$100</td> </tr>
<tr> <td>February</td>
<td>$50</td>
</tr>
</table>
</body>
</html>
```

Sample Output

Table Caption

To add a caption to a table, use the caption tag.

Monthly savings

Month	Savings
January	$100
February	$50

• **A Special Style for One Table**

• To define a special style for one particular table, add an id attribute to the table:

```
<!DOCTYPE html>
<html> <head>
<style> table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
th, td {
padding: 15px; text-align: left;
}
#t01 {
width: 100%; background-color: #f1f1c1;
}
</style> </head> <body><h2>Styling Tables</h2>
<table style="width:100%">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr> <tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td>
<td>Jackson</td> <td>94</td>
</tr> <tr>
<td>John</td> <td>Doe</td>
<td>80</td> </tr>
</table> <br><table id="t01">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr> <tr>
<td>Jill</td> <td>Smith</td>
```

```
<td>50</td> </tr>
<tr> <td>Eve</td>
<td>Jackson</td> <td>94</td>
</tr> <tr>
<td>John</td> <td>Doe</td>
<td>80</td> </tr>
</table>
</body>
</html>
```

Sample Output

Styling Tables

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

- **Add more styles**

```
<!DOCTYPE html>
<html>
<head> <style>
table {
width:100%;
}
table, th, td {
border: 1px solid black;
border-collapse: collapse;
}
th, td {
padding: 15px;
text-align: left;
}#t01 tr:nth-child(even) {
```

```
background-color: #eee;
}
#t01 tr:nth-child(odd) {
background-color: #fff;
}
#t01 th {
background-color: black; color: white;
}
</style> </head>
<body> <h2>Styling Tables</h2>
<table> <tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr><tr>
<td>Jill</td> <td>Smith</td>
<td>50</td> </tr>
<tr> <td>Eve</td>
<td>Jackson</td> <td>94</td>
</tr> <tr>
<td>John</td> <td>Doe</td>
<td>80</td> </tr>
</table> <br>
<table id="t01">
<tr> <th>Firstname</th>
<th>Lastname</th> <th>Age</th>
</tr><tr> <td>Jill</td>
<td>Smith</td> <td>50</td>
</tr> <tr>
<td>Eve</td> <td>Jackson</td>
<td>94</td> </tr>
<tr> <td>John</td>
<td>Doe</td> <td>80</td>
```

</tr> </table>

</body>

</html>

Sample Output

Styling Tables

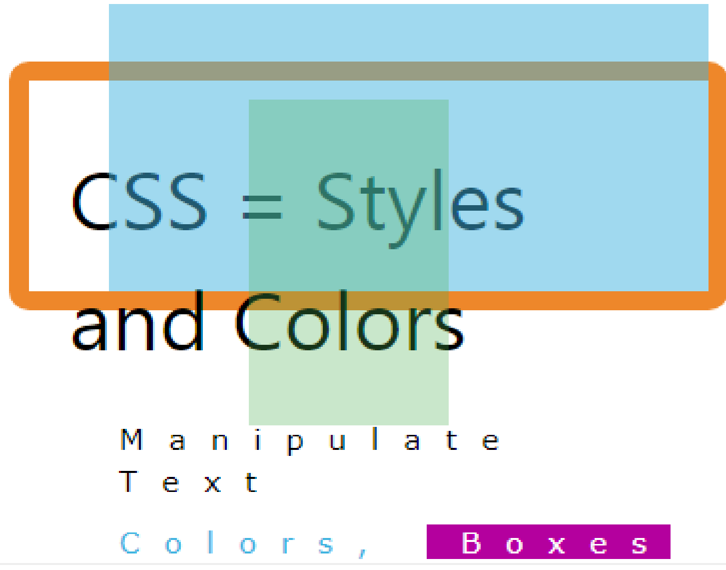
Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

Firstname	Lastname	Age
Jill	Smith	50
Eve	Jackson	94
John	Doe	80

2.15 HTML Styles – CSS

CSS stands for Cascading Style Sheets.

CSS saves a lot of work. It can control the layout of multiple web pages all at once.



- **What is CSS?**
- Cascading Style Sheets (CSS) is used to format the layout of a webpage.
- With CSS, you can control the color, font, the size of text, the spacing between elements, how elements are positioned and laid out, what background images or background colors to be used, different displays for different devices and screen sizes, and much more!
- **Tip:** The word **cascading** means that a style applied to a parent element will also apply to all children elements within the parent. So, if you set the color of the body text to "blue", all headings, paragraphs, and other text elements within the body will also get the same color (unless you specify something else)!
- **Using CSS**
- CSS can be added to HTML documents in 3 ways:
- **Inline** - by using the style attribute inside HTML elements
- **Internal** - by using a <style> element in the <head> section
- **External** - by using a <link> element to link to an external CSS file

• The most common way to add CSS, is to keep the styles in external CSS files.

• **Inline CSS**

• An inline CSS is used to apply a unique style to a single HTML element.

• An inline CSS uses the style attribute of an HTML element.

• The following example sets the text color of the <h1> element to blue, and the text color of the <p> element to red:

• **Example**

• <h1 style="color:blue;">A Blue Heading</h1>

<p style="color:red;">A red paragraph.</p><!DOCTYPE html>

<html> <body>

<h1 style="color:blue;">A Blue Heading</h1>

<p style="color:red;">A red paragraph.</p>

</body> </html>

Sample Output:

A Blue Heading

A red paragraph.

• **Internal CSS**

• An internal CSS is used to define a style for a single HTML page.

• An internal CSS is defined in the <head> section of an HTML page, within a <style> element.

• The following example sets the text color of ALL the <h1> elements (on that page) to blue, and the text color of ALL the <p> elements to red. In addition, the page will be displayed

with a "powderblue" background color:

```
<!DOCTYPE html>
<html>
<head>
<style>
body {background-color: powderblue;}
h1 {color: blue;}
p {color: red;}
</style>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```

Sample Output

This is a heading

This is a paragraph.

• **External CSS**

• An external style sheet is used to define the style for many HTML pages.

• To use an external style sheet, add a link to it in the <head> section of each HTML page:

```
<!DOCTYPE html>
<html> <head>
<link rel="stylesheet" href="styles.css">
</head> <body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body> </html>
```

Sample Output

This is a heading

This is a paragraph.

• The external style sheet can be written in any text editor. The file must not contain any HTML code, and must be saved with a .css extension.

• Here is how the "styles.css" file looks like:

"styles.css":

```
body {
background-color: pink;
}
h1 {
color: blue;
}
p {
color: green;
}
```

• **CSS Colors, Fonts and Sizes**

• Here, we will demonstrate some commonly used CSS properties. You will learn more about them later.

• The CSS color property defines the text color to be used.

• The CSS font-family property defines the font to be used.

• The CSS font-size property defines the text size to be used

.• **Example**

• Use of CSS color, font-family and font-size properties:

```
• <!DOCTYPE html> <html> <head>
<style>
h1 {
color: blue;
font-family: verdana;
font-size: 300%;
}
p {
color: red;
font-family: courier;
font-size: 160%;
```

```
}
</style> </head> <body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body> </html>
```

Sample Output

This is a heading

This is a paragraph.

- **CSS Border**
- The CSS border property defines a border around an HTML element.
- **Tip:** You can define a border for nearly all HTML elements.

- **Example**
- Use of CSS border property:
- p {

border: 2px solid powderblue;

```
}
<!DOCTYPE html>
<html> <head>
<style>
p {
border: 2px solid powderblue;
}
</style> </head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
<p>This is a paragraph.</p>
<p>This is a paragraph.</p>
</body>
</html>
```

Sample Output

This is a heading

This is a paragraph.

This is a paragraph.

This is a paragraph.

• **CSS Padding**

• The CSS padding property defines a padding (space) between the text and the border.

• **Example**

• Use of CSS border and padding properties:

• p {

border: 2px solid powderblue;

padding: 30px;

}•

<!DOCTYPE html>

• <html> <head>

• <style>

• p {

•

border: 2px solid powderblue;

•

padding: 30px;

• }

• </style> </head>

• <body>

• <h1>This is a heading</h1>

• <p>This is a paragraph.</p>

• <p>This is a paragraph.</p>

• <p>This is a paragraph.</p>

• </body> </html>

Sample Output

This is a heading

This is a paragraph.

This is a paragraph.

This is a paragraph.

• **CSS Margin**

• The CSS margin property defines a margin (space) outside the border.

• **Example**

• Use of CSS border and margin properties:

• p {

border: 2px solid powderblue;

margin: 50px;

}• <!DOCTYPE html>

• <html> <head>

• <style>

• p {

• border: 2px solid powderblue;

• margin: 50px;

• }

- </style> </head>
- <body>
- <h1>This is a heading</h1>
- <p>This is a paragraph.</p>
- <p>This is a paragraph.</p>
- <p>This is a paragraph.</p>
- </body> </html>

Sample Output

This is a heading

This is a paragraph.

This is a paragraph.

This is a paragraph.

- **Link to External CSS**
- External style sheets can be referenced with a full URL or with a path relative to the current web page.
- **Example**
- This example uses a full URL to link to a style sheet:
- <link rel="stylesheet" href="https://www.w3sc hools.com/html/styles.css"><!DOCTYPE html>

```
<html>
<head>
<link rel="stylesheet"
href="https://www.w3schools.com/html/styles.css">
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```

Sample Output

This is a heading

This is a paragraph.

- **Example**
- This example links to a style sheet located in the html folder on the current web site:
- <link rel="stylesheet" href="/html/styles.css">

- **Example**
- <!DOCTYPE html>
- <html> <head>
- <link rel="stylesheet" href="/html/styles.css">
- </head>
- <body>
- <h1>This is a heading</h1>
- <p>This is a paragraph.</p>
- </body> </html>

Sample Output

This is a heading

This is a paragraph.

- **Example**
- This example links to a style sheet located in the same folder as the current page:
- <link rel="stylesheet" href="styles.css">

<!DOCTYPE html>

<html>

<head>

```html
<link rel="stylesheet" href="styles.css">
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```

Sample Output

This is a heading

This is a paragraph.

- **Summary**
- Use the HTML style attribute for inline styling
- Use the HTML <style> element to define internal CSS
- Use the HTML <link> element to refer to an external CSS file
- Use the HTML <head> element to store <style> and <link> elements
- Use the CSS color property for text colors
- Use the CSS font-family property for text fonts
- Use the CSS font-size property for text sizes
- Use the CSS border property for borders
- Use the CSS padding property for space inside the border
- Use the CSS margin property for space outside the Border

2.16 Introduction to XML

XML is a software- and hardware-independent tool for storing and transporting data.

What is XML?

• XML stands for eXtensible Markup Language.

•XML is a markup language much like HTML.

•XML was designed to store and transport data.

•XML was designed to be self-descriptive.

•XML is a W3C Recommendation.• **XML Does Not DO Anything**

• Maybe it is a little hard to understand, but XML does not DO anything.

• This note is a note to Tove from Jani, stored as XML:

• <note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Don't forget me this weekend!</body>

</note>

• **The XML above is quite self-descriptive:**

• It has sender information.

• It has receiver information

• It has a heading

• It has a message body.

But still, the XML above does not DO anything. XML is just information wrapped in tags.

Someone must write a piece of software to send, receive, store, or display it:

Note

To: Tove

From: Jani

Reminder

Don't forget me this weekend!

- **The Difference Between XML and HTML**
- XML and HTML were designed with different goals:
- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are.
- **XML Does Not Use Predefined Tags**
- The XML language has no predefined tags.
- The tags in the example above (like <to> and <from>) are not defined in any XML standard.

These tags are "invented" by the author of the XML document.

- HTML works with predefined tags like <p>, <h1>, <table>, etc.
- With XML, the author must define both the tags and the document structure.
- **XML is Extensible**
- Most XML applications will work as expected even if new data is added (or removed).

• Imagine an application designed to display the original version of note.xml (<to> <from> <heading> <body>).

• Then imagine a newer version of note.xml with added <date> and <hour> elements, and a removed <heading>.

• The way XML is constructed, older version of the application can still work:

• <note>
<date>2015-09-01</date>
<hour>08:30</hour>
<to>Tove</to>
<from>Jani</from>
<body>Don't forget me this
weekend!</body>
</note>

Old Version

Note

To: Tove

From: Jani

Reminder

Don't forget me this weekend!

New Version

Note

To: Tove

From: Jani

Date: 2015-09-01 08:30

Don't forget me this weekend!

• **XML Simplifies Things**

• It simplifies data sharing

• It simplifies data transport

• It simplifies platform changes

• It simplifies data availability

• Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems (or upgraded systems) is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.

• XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.

• XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

• With XML, data can be available to all kinds of "reading machines" like people, computers, voice machines, news feeds, etc.

XML is a W3C Recommendation

• XML became a W3C Recommendation as early as in February 1998.

➢ How Can XML be Used?

XML is used in many aspects of web development.

XML is often used to separate data from presentation.

XML Separates Data from Presentation

•XML does not carry any information about how to be displayed.

•The same XML data can be used in many different presentation scenarios.

•Because of this, with XML, there is a full separation between data and presentation.

• **XML is Often a Complement to HTML**

• In many HTML applications, XML is used to store or transport data, while HTML is used to format and display the same data.

• **XML Separates Data from HTML**

• When displaying data in HTML, you should not have to edit the HTML file when the data changes.

• With XML, the data can be stored in separate XML files.

• With a few lines of JavaScript code, you can read an XML file and update the data content of any HTML page.

• Display Books.xml »•

Books.xml

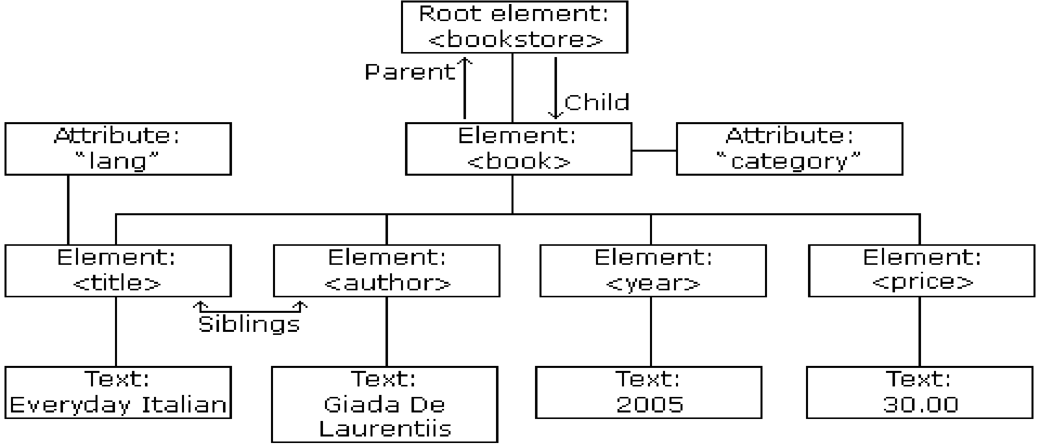
```
• <?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book category="cooking">
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
<book category="children">
<title lang="en">Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book><book category="web">
• <title lang="en">XQuery Kick Start</title>
<author>James McGovern</author>
<author>Per Bothner</author>
<author>Kurt Cagle</author>
<author>James Linn</author>
<author>Vaidyanathan Nagarajan</author>
<year>2003</year>
<price>49.99</price>
</book>
<book category="web" cover="paperback">
• <title lang="en">Learning XML</title>
<author>Erik T. Ray</author>
<year>2003</year>
```

<price>39.95</price>
</book>
</bookstore>

Title	Author
Everyday Italian	Giada De Laurentiis
Harry Potter	J K. Rowling
XQuery Kick Start	James McGovern
Learning XML	Erik T. Ray

➢ XML Tree

XML documents form a tree structure that starts at "the root" and branches to "the leaves".



- **An Example XML Document**
- The image above represents books in this XML:
- <?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book category="cooking">

<title lang="en">Everyday Italian</title>

<author>Giada De Laurentiis</author>

<year>2005</year>

<price>30.00</price>

</book>

<book category="children">

<title lang="en">Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

```
<book category="web">
<title lang="en">Learning XML</title>
<author>Erik T. Ray</author>
<year>2003</year>
<price>39.95</price>
</book>
</bookstore>
```

- **XML Tree Structure**
- XML documents are formed as **element trees**.
- An XML tree starts at a **root element** and branches from the root to **child elements**.
- All elements can have sub elements (child elements):
- <root>

<child>

<subchild>.....</subchild>

</child>

</root>

- The terms parent, child, and sibling are used to describe the relationships between elements.
- Parents have children. Children have parents. Siblings are children on the same level (brothers and sisters).
- All elements can have text content (Harry Potter) and attributes (category="cooking").
- **Self-Describing Syntax**
- XML uses a much self-describing syntax.
- A prolog defines the XML version and the character encoding:
- ***<?xml version="1.0" encoding="UTF-8"?>***
- The next line is the **root element** of the document:
- <bookstore>
- The next line starts a <book> element:
- <book category="cooking">• The <book> elements have **4 child elements**: <title>, <author>, <year>, <price>.
- <title lang="en">Everyday Italian</title>

<author>Giada De Laurentiis</author>

<year>2005</year>

<price>30.00</price>

• The next line ends the book element:

• </book>

• You can assume, from this example, that the XML document contains information about books in a bookstore.

XML Syntax Rules The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.

XML Documents Must Have a Root Element

XML documents must contain one **root** element that is the **parent** of all other elements:

```
<root>
<child>
<subchild>.....</subchild>
</child>
</root>
```

• In this example **<note>** is the root element:

• <?xml version="1.0" encoding="UTF-8"?>

```
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this
weekend!</body>
</note>
```

• **The XML Prolog**

• This line is called the XML **prolog**:

• ***<?xml version="1.0" encoding="UTF-8"?>***

• The XML prolog is optional. If it exists, it must come first in the document.

• XML documents can contain international characters, like Norwegian øæå or French êèé.

• To avoid errors, you should specify the encoding used, or save your XML files as UTF-8.

• UTF-8 is the default character encoding for XML documents.

• UTF-8 is also the default encoding for HTML5, CSS, JavaScript, PHP, and SQL.• All XML Elements Must Have a Closing Tag

• In XML, it is illegal to omit the closing tag. All elements **must** have a closing tag:

• <p>This is a paragraph.</p>

• **Note:** The XML prolog does not have a closing tag! This is not an error. The prolog is not a part of the XML document.

• **XML Tags are Case Sensitive**

• XML tags are case sensitive. The tag <Letter> is different from the tag <letter>.

• Opening and closing tags must be written with the same case:

• <message>This is correct</message>

• "Opening and closing tags" are often referred to as "Start and end tags". Use whatever you prefer. It is exactly the same thing.

• **XML Elements Must be Properly Nested**

• In HTML, you might see improperly nested elements:

• <i>This text is bold and italic</i>

• In XML, all elements **must** be properly nested within each other:

• <i>This text is bold and italic</i>

• In the example above, "Properly nested" simply means that since the <i> element is opened inside the <b> element, it must be closed inside the <b> element.

XML Attribute Values Must Always be Quoted

• XML elements can have attributes in name/value pairs just like in HTML.

• In XML, the attribute values must always be quoted:

• <note date="12/11/2007">
<to>Tove</to>
<from>Jani</from>
</note>

• **Entity References**

• Some characters have a special meaning in XML.

• If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element.

• This will generate an XML error:

• <message>salary < 1000</message>

• To avoid this error, replace the "<" character with an **entity reference**:

• <message>salary < 1000</message>

There are 5 pre-defined entity references in XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Only < and & are strictly illegal in XML, but it is a good habit to replace > with > as well.

• **Comments in XML**

• The syntax for writing comments in XML is similar to that of HTML:

• <!-- This is a comment -->

• Two dashes in the middle of a comment are not allowed:

• <!-- This is an invalid -- comment -->

White-space is Preserved in XML

XML does not truncate multiple white-spaces (HTML truncates multiple white-spaces to one single white-space):

XML:	Hello Tove
HTML:	Hello Tove

XML Stores New Line as LF

Windows applications store a new line as: carriage return and line feed (CR+LF).

Unix and Mac OSX use LF.

Old Mac systems use CR.

XML stores a new line as LF.

• **Well Formed XML**

• XML documents that conform to the syntax rules above are said to be "Well Formed" XML documents.

➢ **XML Elements**

What is an XML Element?

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

<price>29.99</price>

An element can contain:

text

attributes

other elements

or a mix of the above

<bookstore>

```
<book category="children"> <title>Harry Potter</title>
<author>J K. Rowling</author> <year>2005</year>
<price>29.99</price> </book>
<book category="web"> <title>Learning XML</title>
<author>Erik T. Ray</author> <year>2003</year>
<price>39.95</price> </book> </bookstore>
```

- **In the example above:**
- <title>, <author>, <year>, and <price> have **text content** because they contain text (like 29.99).
- <bookstore> and <book> have **element contents**, because they contain elements.
- <book> has an **attribute** (category="children").
- **Empty XML Elements**
- An element with no content is said to be empty.
- In XML, you can indicate an empty element like this:
- <element></element>
- You can also use a so called self-closing tag:
- <element />
- The two forms produce identical results in XML software (Readers, Parsers, Browsers).
- Empty elements can have attributes.
- • **XML Naming Rules**
- XML elements must follow these naming rules:
- Element names are case-sensitive
- Element names must start with a letter or underscore
- Element names cannot start with the letters xml (or XML, or Xml, etc)
- Element names can contain letters, digits, hyphens, underscores, and periods
- Element names cannot contain spaces
- Any name can be used, no words are reserved (except xml).

• **Best Naming Practices**

• Create descriptive names, like this: <person>, <firstname>, <lastname>.

• Create short and simple names, like this: <book_title> not like this: <the_title_of_the_book>.

• Avoid "-". If you name something "first-name", some software may think you want to subtract "name" from "first".

• Avoid ".". If you name something "first.name", some software may think that "name" is a property of the object "first".

• Avoid ":". Colons are reserved for namespaces (more later).

• Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software doesn't support them.

Naming Styles

There are no naming styles defined for XML elements. But here are some commonly used:

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Underscore	<first_name>	Underscore separates words
Pascal case	<FirstName>	Uppercase first letter in each word
Camel case	<firstName>	Uppercase first letter in each word except the first

• If you choose a naming style, it is good to be consistent!

• XML documents often have a corresponding database. A common practice is to use the naming rules of the database for the XML elements.

• Camel case is a common naming rule in JavaScripts.

• **XML Elements are Extensible**

• XML elements can be extended to carry more information.

• Look at the following XML example:

• <note>
<to>Tove</to>
<from>Jani</from>
<body>Don't forget me this weekend!</body>
</note>

• Let's imagine that we created an application that
extracted the <to>, <from>, and <body> elements
from the XML document to produce this output:

• **MESSAGE**

• **To:** Tove
From: Jani

• Don't forget me this weekend!

• Imagine that the author of the XML document added some extra information to it:

• <note>
<date>2008-01-10</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>

• **Should the application break or crash?**

• No. The application should still be able to find the <to>, <from>, and <body> elements in the XML document and produce the same output.

• This is one of the beauties of XML. It can be extended without breaking applications.

- **XML Attributes**

XML elements can have attributes, just like HTML. Attributes are designed to contain data related to a specific element.

XML Attributes Must be Quoted

Attribute values must always be quoted. Either single or double quotes can be used.

For a person's gender, the <person> element can be written like this:

<person gender="female">

or like this:

<person gender='female'>

If the attribute value itself contains double quotes you can use single quotes, like in this example:

<gangster name='George "Shotgun" Ziegler'>

or you can use character entities:

<gangster name="George "Shotgun" Ziegler">

• **XML Elements vs. Attributes**

• Take a look at these examples:

• <person gender="female">

<firstname>Anna</firstname>

<lastname>Smith</lastname>

</person>

• <person>

<gender>female</gender>

<firstname>Anna</firstname>

<lastname>Smith</lastname>

</person>• In the first example gender is an attribute. In the last, gender is an element. Both examples provide the same information.

• There are no rules about when to use attributes or when to use elements in XML.

• **My Favorite Way**

• The following three XML documents contain exactly the same information:

• A date attribute is used in the first example:

• <note date="2008-01-10">

<to>Tove</to>

<from>Jani</from>

</note>• A <date> element is used in the second example:

• <note>

<date>2008-01-10</date>

<to>Tove</to>

<from>Jani</from>

</note>

• An expanded <date> element is used in the third example: (THIS IS MY FAVORITE):

• <note>

<date>

<year>2008</year>

<month>01</month>

<day>10</day>

</date>

<to>Tove</to>

<from>Jani</from>

</note>

• **Avoid XML Attributes?**

• Some things to consider when using attributes are:

• attributes cannot contain multiple values (elements can)

• attributes cannot contain tree structures (elements can)

• attributes are not easily expandable (for future changes)

• Don't end up like this:

• <note day="10" month="01" year="2008"

to="Tove" from="Jani" heading="Reminder"

body="Don't forget me this weekend!">

</note>

• XML Attributes for Metadata

• Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. This example demonstrates this:

• <messages>
<note id="501">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
<note id="502">
<to>Jani</to>
<from>Tove</from>
<heading>Re: Reminder</heading>
<body>I will not</body>
</note>
</messages>

• The id attributes above are for identifying the different notes. It is not a part of the note itself.

• What I'm trying to say here is that metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.

➢ **XML Namespaces**

XML Namespaces provide a method to avoid element name conflicts.

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications. This XML carries HTML table information:

<table>
<tr>
<td>Apples</td>

<td>Bananas</td>

</tr>

</table>• This XML carries information about a table (a piece of furniture):

• <table>

<name>African Coffee Table</name>

<width>80</width>

<length>120</length>

</table>

• If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

• A user or an XML application will not know how to handle these differences.

• **Solving the Name Conflict Using a Prefix**

• Name conflicts in XML can easily be avoided using a name prefix.

• This XML carries information about an HTML table, and a piece of furniture:

• <h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

• In the example above, there will be no conflict because the two
<table> elements have different names.

• **XML Namespaces - The xmlns Attribute**

• When using prefixes in XML, a **namespace** for the prefix must be defined.

• The namespace can be defined by an **xmlns** attribute in the start tag of an element.

• The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

• <root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

</root>

• **In the example above:**

• The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.

• The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.

• When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

• Namespaces can also be declared in the XML root element:

• <root xmlns:h="http://www.w3.org/TR/html4/"

xmlns:f="https://www.w3schools.com/furniture">

<h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

</root>

• **Note:** The namespace URI is not used by the parser to look up information.

• The purpose of using an URI is to give the namespace a unique name.

• However, companies often use the namespace as a pointer to a web page containing namespace information.

• **Uniform Resource Identifier (URI)**

• A **Uniform Resource Identifier** (URI) is a string of characters which identifies an Internet Resource.

• The most common URI is the **Uniform Resource Locator** (URL) which identifies an Internet domain address. Another, not so common type of URI is the **Uniform Resource Name** (URN).

• **Default Namespaces**

• Defining a default namespace for an element saves us from using prefixes in all the child elements. It has the following syntax:

• xmlns="*namespaceURI*"

• This XML carries HTML table information:

• <table xmlns="http://www.w3.org/TR/html4/">

<tr>

<td>Apples</td>

<td>Bananas</td>

</tr>

</table>• This XML carries information about a piece of furniture:

• <table xmlns="https://www.w3schools.com/furniture">

<name>African Coffee Table</name>

<width>80</width>

```
<length>120</length>
</table>
```

• **Namespaces in Real Use**

• XSLT is a language that can be used to transform XML documents into other formats.

• The XML document below, is a document used to transform XML into HTML.

• The namespace "http://www.w3.org/1999/XSL/Transform" identifies XSLT elements inside an HTML document:

```
• <?xml version="1.0" encoding="UTF-8"?> <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr>
<th style="text-align:left">Title</th>
<th style="text-align:left">Artist</th>
</tr>• <xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

2.17 XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

What is a DTD?

DTD stands for Document Type Definition.

A DTD defines the structure and the legal elements and attributes of an XML document

.• **Valid XML Documents**

• A "Valid" XML document is "Well Formed", as well as it conforms to the rules of a DTD:

```
• <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

• The DOCTYPE declaration above contains a reference to a DTD file. The content of the DTD file is shown and explained below.

• **XML DTD**

• The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:

• **Note.dtd:**

```
• <!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

- **The DTD above is interpreted like this:**
- !DOCTYPE note - Defines that the root element of the document is note
- !ELEMENT note - Defines that the note element must contain the elements: "to, from, heading, body"
- !ELEMENT to - Defines the to element to be of type "#PCDATA"
- !ELEMENT from - Defines the from element to be of type "#PCDATA"
- !ELEMENT heading - Defines the heading element to be of type "#PCDATA"
- !ELEMENT body - Defines the body element to be of type "#PCDATA"
- **Tip:** #PCDATA means parseable character data
- **Using DTD for Entity Declaration**
- A DOCTYPE declaration can also be used to define special characters or strings, used in the document:
- **Example**
-
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note [
<!ENTITY nbsp " ">
<!ENTITY writer "Writer: Donald Duck.">
<!ENTITY copyright "Copyright: W3Schools.">
]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
<footer>&writer; &copyright;</footer>
</note>
```

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
  <footer>Writer: Donald Duck. Copyright: W3Schools.</footer>
</note>
```

Tip: An entity has three parts: it starts with an ampersand (&), then comes the entity name, and it ends with a semicolon (;).

• **When to Use a DTD?**

• With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.

• With a DTD, you can verify that the data you receive from the outside world is valid.

• You can also use a DTD to verify your own data.

• **When NOT to Use a DTD?**

• XML does not require a DTD.

• When you are experimenting with XML, or when you are working with small XML files, creating DTDs may be a waste of time.

• If you develop applications, wait until the specification is stable before you add a DTD. Otherwise, your software might stop working because of validation errors.

•**XML Schema**

An XML Schema describes the structure of an XML document, just like a DTD.

An XML document with correct syntax is called "Well Formed".

An XML document validated against an XML Schema is both "Well Formed" and "Valid".•

2.18 XML Schema

• XML Schema is an XML-based alternative to DTD:

```
• <xs:element name="note">
<xs:complexType>
<xs:sequence>
<xs:element name="to" type="xs:string"/>
<xs:element name="from" type="xs:string"/>
<xs:element name="heading" type="xs:string"/>
<xs:element name="body" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

• **The Schema above is interpreted like this:**

• <xs:element name="note"> defines the element called "note"

• <xs:complexType> the "note" element is a complex type

• <xs:sequence> the complex type is a sequence of elements

• <xs:element name="to" type="xs:string"> the element "to" is of type string (text)

• <xs:element name="from" type="xs:string"> the element "from" is of type string

• <xs:element name="heading" type="xs:string"> the element "heading" is of type string

• <xs:element name="body" type="xs:string"> the element "body" is of type string

• **XML Schemas are More Powerful than DTD**

• XML Schemas are written in XML

• XML Schemas are extensible to additions

• XML Schemas support data types

• XML Schemas support namespaces

• **Why Use an XML Schema?**

• With XML Schema, your XML files can carry a description of its own format.

• With XML Schema, independent groups of people can agree on a standard for interchanging data.

• With XML Schema, you can verify data.• **XML Schemas Support Data Types**

• One of the greatest strengths of XML Schemas is the support for data types:

• It is easier to describe document content

• It is easier to define restrictions on data

• It is easier to validate the correctness of data

• It is easier to convert data between different data types

• **XML Schemas use XML Syntax**

• Another great strength about XML Schemas is that they are written in XML:

• You don't have to learn a new language

• You can use your XML editor to edit your Schema files

• You can use your XML parser to parse your Schema files

• You can manipulate your Schemas with the XML DOM

• You can transform your Schemas with XSLT

2.19 XML Parser

All major browsers have a built-in XML parser to access and manipulate XML.

XML Parser

The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.

However, before an XML document can be accessed, it must be loaded into an XML DOM object. All modern browsers have a built-in XML parser that can convert text into an XML DOM object

.• **Parsing a Text String**

• This example parses a text string into an XML DOM object, and extracts the info from it with JavaScript:

• **Example**

```
<!DOCTYPE html>
<html> <body>
<p id="demo"></p>
<script>
var parser, xmlDoc;
var text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
```

```
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";
parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");
document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script> </body>
</html>
```

Everyday Italian

- **Example Explained**
- A text string is defined:
- text = "<bookstore><book>" +

"<title>Everyday Italian</title>" +

"<author>Giada De Laurentiis</author>" +

"<year>2005</year>" +

```
"</book></bookstore>";
```

- **An XML DOM parser is created:**
- parser = new DOMParser();
- The parser creates a new XML DOM object using the text string:
- xmlDoc = parser.parseFromString(text,"text/xml");
- **Old Versions of Internet Explorer**
- Old versions of Internet Explorer (IE5, IE6, IE7, IE8) do not support the DOMParser object.
- To handle older versions of Internet Explorer, check if the browser supports the DOMParser object, or else create an ActiveXObject:

```
<!DOCTYPE html>
<html> <body>
<p id="demo"></p>
<script>
var parser, xmlDoc;
var text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";
if (window.DOMParser) {
parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");
} else {
xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async = false;
xmlDoc.loadXML(text);
}
document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script> </body> </html>
```

Everyday Italian

- **The XMLHttpRequest Object**
- The XMLHttpRequest Object has a built in XML Parser.
- The **responseText** property returns the response as a string.
- The **responseXML** property returns the response as an XML DOM object.
- If you want to use the response as an XML DOM object, you can use the responseXML property.

```
<!DOCTYPE html>
<html> <body>
<h2>My CD Collection:</h2>
<button type="button" onclick="loadXMLDoc()">
Get my CD collection</button>
<p id="demo"></p>
<script>
function loadXMLDoc() {
var xmlhttp = new XMLHttpRequest();
```

```
xmlhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
myFunction(this);
}
};
xmlhttp.open("GET", "cd_catalog.xml", true);
xmlhttp.send();
}function myFunction(xml) {
var x, i, xmlDoc, txt;
xmlDoc = xml.responseXML;
txt = "";
x = xmlDoc.getElementsByTagName("ARTIST");
for (i = 0; i< x.length; i++) {
txt += x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
}
</script>
</body>
</html>
```

My CD Collection:

Get my CD collection

My CD Collection:

Get my CD collection

Bob Dylan
Bonnie Tyler
Dolly Parton
Gary Moore
Eros Ramazzotti
Bee Gees
Dr.Hook
Rod Stewart
Andrea Bocelli
Percy Sledge
Savage Rose
Many
Kenny Rogers

Will Smith
Van Morrison
Jorn Hoel
Cat Stevens
Sam Brown
T'Pau
Tina Turner
Kim Larsen
Luciano Pavarotti
Otis Redding
Simply Red
The Communards
Joe Cocker

- **XML Parsers**
- An XML parser is a software library or package that provides interfaces for client applications to work with an XML document. The XML Parser is designed to read the XML and create a way for programs to use XML.
- XML parser validates the document and check that the document is well formatted.

• **Types of XML Parsers**

• These are the two main types of XML Parsers:

• DOM

• SAX

• **DOM (Document Object Model)**

• A DOM document is an object which contains all the information of an XML document. It is composed like a tree structure. The DOM Parser implements a DOM API. This API is very simple to use.

• **Features of DOM Parser**

• A DOM Parser creates an internal structure in memory which is a DOM document object and the client applications get information of the original XML document by invoking methods on this document object.

• DOM Parser has a tree based structure.

• **Advantages**

• 1) It supports both read and write operations and the API is very simple to use.

• 2) It is preferred when random access to widely separated parts of a document is required.

• **Disadvantages**

• 1) It is memory inefficient. (consumes more memory because the whole XML document needs to loaded into memory).

• 2) It is comparatively slower than other parsers.

• **SAX (Simple API for XML)**

• A SAX Parser implements SAX API. This API is an event based API and less intuitive.

• **Features of SAX Parser**

• It does not create any internal structure.

• Clients does not know what methods to call, they just overrides the methods of the API and place his own code inside method.

• It is an event based parser, it works like an event handler in Java.

• **Advantages**

• 1) It is simple and memory efficient.

• 2) It is very fast and works for huge documents.

• **Disadvantages**

• 1) It is event-based so its API is less intuitive.

• 2) Clients never know the full information because the data is broken into pieces.

XML DOM

Root element: <bookstore>
Parent
Child
Attribute: "lang"
Element: <book>
Attribute: "category"
Element: <title>
Element: <author>
Element: <year>
Element: <price>
Siblings
Text: Everyday Italian
Text: Giada De Laurentiis
Text: 2005
Text: 30.00

• **What is the DOM?**

• The DOM defines a standard for accessing and manipulating documents:

• *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."*

• The HTML DOM defines a standard way for accessing and manipulating HTML documents. It presents an HTML document as a tree-structure.

• The XML DOM defines a standard way for accessing and manipulating XML documents. It presents an XML document as a tree-structure.

• Understanding the DOM is a must for anyone working with HTML or XML.

• **The HTML DOM** • All HTML elements can be accessed through the HTML DOM.

• This example changes the value of an HTML element with id="demo":

```
<!DOCTYPE html>
<html>
<body>
<h1 id="demo">This is a Heading</h1>
<button type="button"
onclick="document.getElementById('demo').innerHTML = 'Hello World!'">Click Me!
</button>
</body>
</html>
```

Sample Output

This is a Heading

Click Me!

Hello World!

Click Me!

• **The XML DOM**

• All XML elements can be accessed through the XML DOM.

• **Books.xml**

• <?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book category="cooking">

<title lang="en">Everyday Italian</title>

<author>Giada De Laurentiis</author>

<year>2005</year>

<price>30.00</price>

</book>

<book category="children">

<title lang="en">Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

```
<price>29.99</price>
</book>
</bookstore>
```

• This code retrieves the text value of the first <title> element in an XML document:

• **Example**

• txt =
```
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
```

• The XML DOM is a standard for how to get, change, add, and delete XML elements.

• This example loads a text string into an XMLDOM object, and extracts the info from it with JavaScript

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
<script>
var parser, xmlDoc;
var text = "<bookstore><book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book></bookstore>";
parser = new DOMParser();
xmlDoc = parser.parseFromString(text,"text/xml");
document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
</script>
</body>
</html>
```

Sample Output

Everyday Italian

• **The XML DOM**

• All XML elements can be accessed through the XML DOM.

• The XML DOM is:

• A standard object model for XML

• A standard programming interface for XML

• Platform- and language-independent

• A W3C standard

• In other words: **The XML DOM is a standard for how to get, change, add, or delete XML elements.• Get the Value of an XML Element**

• This code retrieves the text value of the first <title> element in an XML document:

• **Example**

• txt = xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;

• **Loading an XML File**

• The XML file used in the examples below is books.xml.

• This example reads "books.xml" into xmlDoc and retrieves the text value of the first <title> element in books.xml:

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
myFunction(this);
}
};
xhttp.open("GET", "books.xml", true);
xhttp.send();
function myFunction(xml) {
var xmlDoc = xml.responseXML;
document.getElementById("demo").innerHTML =
xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
}
</script>
</body>
</html>
```

Sample Output

Everyday Italian

- **Example Explained**
- **xmlDoc** - the XML DOM object created by the parser.
- **getElementsByTagName("title")[0]** - get the first <title> element
- **childNodes[0]** - the first child of the <title> element (the text node)
- **nodeValue** - the value of the node (the text itself)
- Loading an XML String
- This example loads a text string into an XML DOM object, and extracts the info from it with JavaScript:

- **Programming Interface**
- The DOM models XML as a set of node objects. The nodes can be accessed with JavaScript or other programming languages. In this tutorial we use JavaScript.
- The programming interface to the DOM is defined by a set standard properties and methods.
- **Properties** are often referred to as something that is (i.e. nodename is "book").
- **Methods** are often referred to as something that is done (i.e. delete "book").

• **XML DOM Properties**

• These are some typical DOM properties:

• x.nodeName - the name of x

• x.nodeValue - the value of x

• x.parentNode - the parent node of x

• x.childNodes- the child nodes of x

• x.attributes - the attributes nodes of x

• Note: In the list above, x is a node object.

• **XML DOM Methods**

• x.getElementsByTagName(*name*) - get all elements with a specified tag name

• x.appendChild(*node*) - insert a child node to x

• x.removeChild(*node*) - remove a child node from x

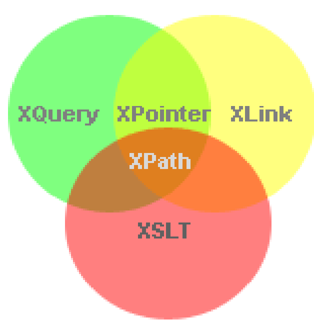
• Note: In the list above, x is a node object.**XPath**

2.20 What is XPath?

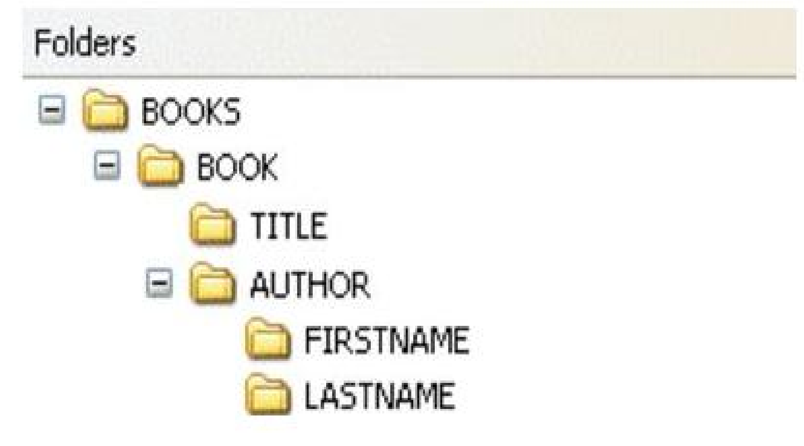
XPath is a major element in the XSLT standard.

XPath can be used to navigate through elements

and attributes in an XML document.



- XPath stands for XML Path Language
- XPath uses "path like" syntax to identify and navigate nodes in an XML document
- XPath contains over 200 built-in functions
- XPath is a major element in the XSLT standard
- XPath is a W3C recommendation



- **XPath Standard Functions**
- XPath includes over 200 built-in functions.
- There are functions for string values, numeric values, booleans, date and time comparison, node manipulation, sequence manipulation, and much more.
- Today XPath expressions can also be used in JavaScript, Java, XML Schema, PHP, Python, C and C++, and lots of other languages.

- **XPath is Used in XSLT**
- XPath is a major element in the XSLT standard.
- With XPath knowledge you will be able to take great advantage of your XSLT knowledge.
- **XPath is a W3C Recommendation**
- XPath 1.0 became a W3C Recommendation on November 16, 1999.
- XPath 2.0 became a W3C Recommendation on January 23, 2007.

• XPath 3.0 became a W3C Recommendation on April 8, 2014

• **XPath Nodes**

• **XPath Terminology**

• **Nodes**

• In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes.

• XML documents are treated as trees of nodes. The topmost element of the tree is called the root element.

• **Look at the following XML document:**

```
• <?xml version="1.0" encoding="UTF-8"?>
<bookstore>
<book>
<title lang="en">Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
</bookstore>
```

• **Example of nodes in the XML document above:**

• <bookstore> (root element node)

<author>J K. Rowling</author> (element node)

lang="en" (attribute node)

• **Atomic values**

• Atomic values are nodes with no children or parent.

• **Example of atomic values:**

• J K. Rowling

"en"

• Items

• Items are atomic values or nodes.

- **Relationship of Nodes**
- **Parent**
- Each element and attribute has one parent.
- In the following example; the book element is the parent of the title,
author, year, and price:
- <book>
<title>Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
- **Children**
- Element nodes may have zero, one or more children.
- In the following example; the title, author, year, and price elements are all children of the book
element:
- <book>
<title>Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
- **Siblings**
- Nodes that have the same parent.
- In the following example; the title, author, year, and price elements are all siblings:
- <book>
<title>Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>

• **Ancestors**
• A node's parent, parent's parent, etc.
• In the following example; the ancestors of the title element are the book element and the bookstore
element:
• <bookstore>
<book>
<title>Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
</bookstore>
• **Descendants**
• A node's children, children's children, etc.
• In the following example; descendants of the bookstore element are the book, title, author, year, and price elements:
• <bookstore>
<book>
<title>Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
</bookstore>

• **XPath Syntax**
• XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps.
• **The XML Example Document**
• We will use the following XML document in the examples below.

• <?xml version="1.0" encoding="UTF-8"?>

<bookstore>

<book>

<title lang="en">Harry Potter</title>

<price>29.99</price>

</book>

<book>

<title lang="en">Learning XML</title>

<price>39.95</price>

</book>

</bookstore>

Selecting Nodes

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

Predicates

Predicates are used to find a specific node or a node that contains a specific value.

Predicates are always embedded in square brackets.

In the table below we have listed some path expressions with predicates and the result of the expressions:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. **Note:** In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath:

	In JavaScript: *xml*.setProperty("SelectionLanguage","XPath");
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML nodes.

Wildcard	**Description**
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
/bookstore/*	Selects all the child element nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have at least one attribute of any kind

Selecting Several Paths

By using the | operator in an XPath expression you can select several paths.

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document
/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

2.21 XPath Operators

An XPath expression returns either a node-set, a string, a Boolean, or a number.

XPath Operators

Below is a list of the operators that can be used in XPath expressions:

Operator	Description	Example
|	Computes two node-sets	//book | //cd
+	Addition	6 + 4
-	Subtraction	6 - 4
*	Multiplication	6 * 4
div	Division	8 div 4
=	Equal	price=9.80
!=	Not equal	price!=9.80
<	Less than	price<9.80
<=	Less than or equal to	price<=9.80
>	Greater than	price>9.80
>=	Greater than or equal to	price>=9.80
or	or	price=9.80 or price=9.70
and	and	price>9.00 and price<9.90
mod	Modulus (division remainder)	5 mod 2

2.22 XSLT Introduction

XSL (eXtensible Stylesheet Language) is a styling language for XML.

XSLT stands for XSL Transformations.

This tutorial will teach you how to use XSLT to transform XML documents into other formats (like transforming XML into HTML).

Online XSLT Editor

With our online editor, you can edit XML and XSLT code, and click on a button to view the result.

•
```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
</tr>
<xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

• **What You Should Already Know**

• Before you continue you should have a basic understanding of the following:

• HTML

• XML• **XSL(T) Languages**

• **XSLT** is a language for transforming XML documents.

• **XPath** is a language for navigating in XML documents.

• **XQuery** is a language for querying XML documents.

• **It Started with XSL**

• XSL stands for E**X**tensible **S**tylesheet **L**anguage.

• The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Stylesheet Language

.• **CSS = Style Sheets for HTML**

• HTML uses predefined tags. The meaning of, and how to display each tag is well understood.

• CSS is used to add styles to HTML elements.

• **XSL = Style Sheets for XML**

• XML does not use predefined tags, and therefore the meaning of each tag is not well understood.

• A <table> element could indicate an HTML table, a piece of furniture, or something else - and browsers do not know how to display it!

• So, XSL describes how the XML elements should be displayed.

• **XSL - More Than a Style Sheet Language**

• XSL consists of four parts:

• XSLT - a language for transforming XML documents

• XPath - a language for navigating in XML documents

• XSL-FO - a language for formatting XML documents (discontinued in 2013)

• XQuery - a language for querying XML documents

• With the **CSS3 Paged Media Module**, W3C has delivered a new standard for document formatting. So, since 2013, CSS3 is proposed as an XSL-FO replacement.

• **What is XSLT?**
• XSLT stands for XSL Transformations
• XSLT is the most important part of XSL
• XSLT transforms an XML document into another XML document
• XSLT uses XPath to navigate in XML documents
• XSLT is a W3C Recommendation• **XSLT = XSL Transformations**
• XSLT is the most important part of XSL.
• XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML.
Normally XSLT does this by transforming each XML element into an (X)HTML element.
• With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.
• A common way to describe the transformation process is to say that **XSLT transforms an XML source-tree into an XML result-tree**.

• **XSLT Uses XPath**
• XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.
• **XSLT - Transformation**
• **Correct Style Sheet Declaration**
• The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>.
• **Note:** <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used!
• The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:
• <xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
• or:
• <xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

• To get access to the XSLT elements, attributes and features we must declare the XSLT namespace at the top of the document.
• The xmlns:xsl="http://www.w3.org/1999/XSL/Transform" points to the official W3C XSLT namespace. If you use this namespace, you must also include
the attribute version="1.0".

• **Start with a Raw XML Document**
• We want to **transform** the following XML document ("cdcatalog.xml") into XHTML:
• <?xml version="1.0" encoding="UTF-8"?>
<catalog>
<cd>
<title>Empire Burlesque</title>
<artist>Bob Dylan</artist>
<country>USA</country>
<company>Columbia</company>
<price>10.90</price>
<year>1985</year>
</cd>
.
.
</catalog>

• **XSLT <xsl:template> Element**
• An XSL style sheet consists of one or more set of rules that are called templates.
• A template contains rules to apply when a specified node is matched.
• **The <xsl:template> Element**
• The <xsl:template> element is used to build templates.
• The **match** attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

- **XSLT <xsl:value-of> Element**
- The <xsl:value-of> element is used to extract the value of a selected node.
- The <xsl:value-of> Element
- The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output <?xstream of the transformation:
- **Example**

```
• xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
</tr>
<tr>
<td><xsl:value-of select="catalog/cd/title"/></td>
<td><xsl:value-of select="catalog/cd/artist"/></td>
</tr>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>• XSLT <xsl:for-each> Element
```

• The <xsl:for-each> element allows you to do looping in XSLT.

• The <xsl:for-each> Element

• The XSL <xsl:for-each> element can be used to select every XML element of a specified node-set:

• Example

```
• <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">• <tr bgcolor="#9acd32">
```

```
<th>Title</th>
<th>Artist</th>
</tr>
<xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

• **Filtering the Output**

• We can also filter the output from the XML file by adding a criterion to the select attribute in the
<xsl:for-each> element.

• **<xsl:for-each select="catalog/cd[artist='Bob Dylan']">**

• Legal filter operators are:

• = (equal)

• != (not equal)

• < less than

• > greater than

• Take a look at the adjusted XSL style sheet:• <?xml version="1.0" encoding="UTF-8"?>

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
```

```
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
</tr>
<xsl:for-each select="catalog/cd[artist='Bob Dylan']">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template> </xsl:stylesheet>• XSLT <xsl:sort> Element
```

• The <xsl:sort> element is used to sort the output.

• Where to put the Sort Information

• To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file:

• **Example**

```
• <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
```

```
<th>Artist</th>
</tr>• <xsl:for-each select="catalog/cd">
<xsl:sort select="artist"/>
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

• **Note:** The **select** attribute indicates what XML element to sort on.

• **XSLT <xsl:if> Element**

• The <xsl:if> element is used to put a conditional test against the content of the XML file.

• The <xsl:if> Element

• To put a conditional if test against the content of the XML file, add an <xsl:if> element to the XSL document.

• **Syntax**

• <xsl:if test="*expression*">
...some output if the expression is true...
</xsl:if>

• **Where to Put the <xsl:if> Element**

• To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

• **Example**

• <?xml version="1.0" encoding="UTF-8"?>

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
<th>Price</th>
</tr>
<xsl:for-each select="catalog/cd">
<xsl:if test="price > 10">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
<td><xsl:value-of select="price"/></td>
</tr>
</xsl:if>
</xsl:for-each> </table> </body> </html>
</xsl:template> </xsl:stylesheet>
```

• **XSLT <xsl:choose> Element**

• The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests.

• The <xsl:choose> Element

• Syntax

• <xsl:choose>

<xsl:when test="*expression*">

... some output ...

```
</xsl:when>
<xsl:otherwise>
... some output ....
</xsl:otherwise>
</xsl:choose>
```

• **Where to put the Choose Condition**

• To insert a multiple conditional test against the XML file, add the <xsl:choose>, <xsl:when>, and <xsl:otherwise> elements to the XSL file:

• **Example**

```
• <?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html> <body>
<h2>My CD Collection</h2>
<table border="1">
<tr bgcolor="#9acd32">
<th>Title</th>
<th>Artist</th>
</tr> <xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<xsl:choose>
<xsl:when test="price > 10">
<td bgcolor="#ff00ff">
<xsl:value-of select="artist"/></td>
</xsl:when>
<xsl:otherwise>
<td><xsl:value-of select="artist"/></td>
</xsl:otherwise>
</xsl:choose> </tr>
```

```
</xsl:for-each> </table> </body> </html>
</xsl:template> </xsl:stylesheet>
```

• **XSLT <xsl:apply-templates> Element**

• The <xsl:apply-templates> element applies a template rule to the current element or to the current element's child nodes.

• The <xsl:apply-templates> Element

• The <xsl:apply-templates> element applies a template to the current element or to the current element's child nodes.

• If we add a "select" attribute to the <xsl:apply templates> element, it will process only the child elements that matches the value of the attribute.

We can use the "select" attribute to specify in which order the child nodes are to be processed.•

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html> <body>
<h2>My CD Collection</h2>
<xsl:apply-templates/>
</body> </html>
</xsl:template>
<xsl:template match="cd">
<p> <xsl:apply-templatesselect="title"/>
<xsl:apply-templates select="artist"/>
</p> </xsl:template>
<xsl:template match="title">
Title: <span style="color:#ff0000">
<xsl:value-ofselect="."/></span>
<br /> </xsl:template> <xsl:template match="artist">
Artist: <span style="color:#00ff00">
<xsl:value-ofselect="."/></span>
```

```
<br /> </xsl:template>
</xsl:stylesheet>
```

XSLT Elements

The links in the "Element" column point to attributes and more useful information about each specific element.

Element	Description
apply-imports	Applies a template rule from an imported style sheet
apply-templates	Applies a template rule to the current element or to the current element's child nodes
attribute	Adds an attribute
attribute-set	Defines a named set of attributes
call-template	Calls a named template
choose	Used in conjunction with <when> and <otherwise> to express multiple conditional tests
comment	Creates a comment node in the result tree
copy	Creates a copy of the current node (without child nodes and attributes)
copy-of	Creates a copy of the current node (with child nodes and attributes)

decimal-format	Defines the characters and symbols to be used when converting numbers into strings, with the format-number() function
element	Creates an element node in the output document
fallback	Specifies an alternate code to run if the processor does not support an XSLT element
for-each	Loops through each node in a specified node set
if	Contains a template that will be applied only if a specified condition is true
import	Imports the contents of one style sheet into another. **Note:** An imported style sheet has lower precedence than the importing style sheet
include	Includes the contents of one style sheet into another. **Note:** An included style sheet has the same precedence as the including style sheet
key	Declares a named key that can be used in the style sheet with the key() function
message	Writes a message to the output (used to report errors)
namespace-alias	Replaces a namespace in the style sheet to a different namespace in the output

number	Determines the integer position of the current node and formats a number
otherwise	Specifies a default action for the <choose> element
output	Defines the format of the output document
param	Declares a local or global parameter
preserve-space	Defines the elements for which white space should be preserved
processing-instruction	Writes a processing instruction to the output
sort	Sorts the output
strip-space	Defines the elements for which white space should be removed
stylesheet	Defines the root element of a style sheet
template	Rules to apply when a specified node is matched
text	Writes literal text to the output

transform	Defines the root element of a style sheet
value-of	Extracts the value of a selected node
variable	Declares a local or global variable
when	Specifies an action for the <choose> element
with-param	Defines the value of a parameter to be passed into a template

UNIT-III

JAVA SCRIPT: Introduction to JavaScript, Declaring Java Script Variables, Basics of JavaScript, Control Structures, Pop Up Boxes, Functions, Arrays, Events, Objects.

JavaScript

• JavaScript is the world's most popular programming language.

• JavaScript is the programming language of the Web.

• JavaScript is easy to learn.

3.1 JavaScript Introduction

•JavaScript Can Change HTML Content.One of many JavaScript HTML methods is getElementById().

•The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript":<!DOCTYPE html>

```
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p id="demo">JavaScript can change HTML content.</p>
<button type="button" onclick='document.getElementById("demo"). innerHTML = "Hello
JavaScript!"'>Click Me!</button>
</body>
</html>
```

Sample Output

What Can JavaScript Do?

JavaScript can change HTML content.

Click Me!

What Can JavaScript Do?

Hello JavaScript!

Click Me!

• JavaScript accepts both double and single quotes:

```
<!DOCTYPE html>
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p id="demo">JavaScript can change HTML
content.</p>
<button type="button"
onclick="document.getElementById('demo').inne
rHTML = 'Hello JavaScript!'">Click Me!</button>
</body>
</html>
```

Sample Output

What Can JavaScript Do?

JavaScript can change HTML content.

Click Me!

What Can JavaScript Do?

Hello JavaScript!

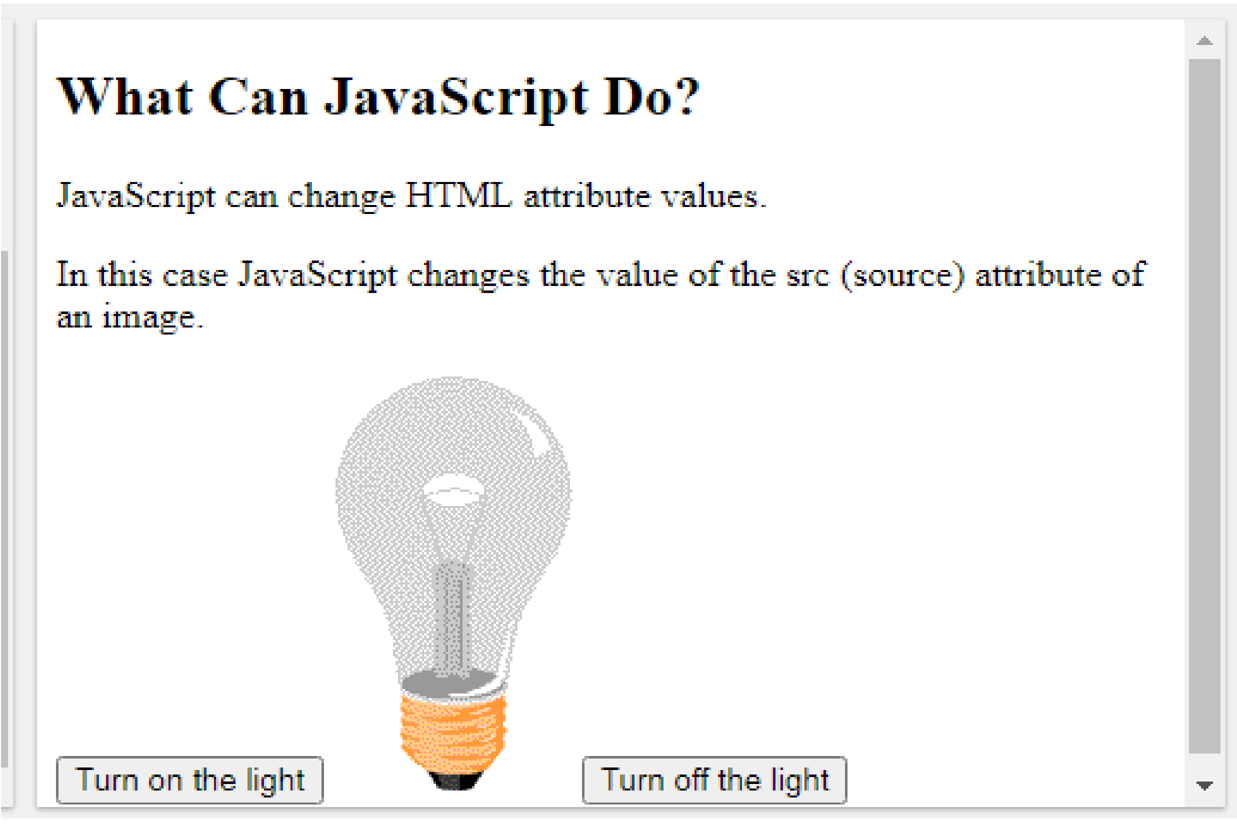
Click Me!

- **JavaScript Can Change HTML Attribute Values**
- In this example JavaScript changes the value of the src (source) attribute of an <img> tag:

```
<!DOCTYPE html>
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p>JavaScript can change HTML attribute values.</p>
<p>In this case JavaScript changes the value of the src (source)
attribute of an image.</p>
<button onclick="document.getElementById('myImage').src='pic_bulbon.gif' ">Turn on the
light</button>
```

```
<img id="myImage" src="pic_bulboff.gif" style="width:100px">
<button onclick="document.getElementById('myImage').src='pic_bulboff.gif' ">Turn off the
light</button>
</body>
</html>
```

The Light Bulb



• **JavaScript Can Change HTML Styles (CSS)**

• Changing the style of an HTML element, is a variant of changing an HTML attribute:

```
<!DOCTYPE html>
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p id="demo">JavaScript can change the style of an
HTML element.</p>
<button type="button" onclick="document.getElementById('demo').style.f
```

```
ontSize='35px'">Click Me!</button>
</body>
</html>
```

Sample Output

What Can JavaScript Do?

JavaScript can change the style of an HTML element.

Click Me!

What Can JavaScript Do?

JavaScript can change the style of an HTML element.

Click Me!

- **JavaScript Can Hide HTML Elements**
- Hiding HTML elements can be done by changing the display style:

```
<!DOCTYPE html>
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p id="demo">JavaScript can hide HTML
elements.</p>
```

```
<button type="button"
onclick="document.getElementById('demo').style
.display='none'">Click Me!</button>
</body>
</html>
```

Sample Output

What Can JavaScript Do?

JavaScript can hide HTML elements.

Click Me!

What Can JavaScript Do?

Click Me!

• **JavaScript Can Show HTML Elements**

• Showing hidden HTML elements can also be done by changing the display style:

```
<!DOCTYPE html>
<html>
<body>
<h2>What Can JavaScript Do?</h2>
<p>JavaScript can show hidden HTML elements.</p>
• <p id="demo" style="display:none">Hello JavaScript!</p>
<button type="button" onclick="document.getElementById('demo').style.display='
```

block'">Click Me!</button>

</body>

</html>

Sample Output

What Can JavaScript Do?

JavaScript can show hidden HTML elements.

Click Me!

What Can JavaScript Do?

JavaScript can show hidden HTML elements.

Hello JavaScript!

Click Me!

JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

Writing into an HTML element, using innerHTML.

Writing into the HTML output using document.write( ).

Writing into an alert box, using window.alert().

Writing into the browser console, using console.log().

• **Using innerHTML**

• To access an HTML element, JavaScript can use the document.getElementById(id) method.

• The id attribute defines the HTML element. The innerHTML property defines the HTML content:

• **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Web Page</h2>
<p>My First Paragraph.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

Sample Output

My First Web Page

My First Paragraph.

11

• **Using document.write()**

• For testing purposes, it is convenient to use document.write():

Example:

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Web Page</h2>
<p>My first paragraph.</p>
```

```
<p>Never call document.write after the document has finished loading.
It will overwrite the whole document.</p>
<script>
document.write(5 + 6);
</script>
</body>
</html>
```

Sample Output

My First Web Page

My first paragraph.

Never call document.write after the document has finished loading. It will overwrite the whole document.

11

- Using document.write() after an HTML document is loaded, will **delete all existing HT**ML:
- **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>My First Web Page</h2>
<p>My first paragraph.</p>
<button type="button" onclick="document.write(5 + 6)">Try it</button>
</body>
</html>
```

Sample Output

My First Web Page

My first paragraph.

Try it

11

- **Using window.alert()**
- You can use an alert box to display data:

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
window.alert(5 + 6);
</script>
</body>
</html>
```

Sample Output

An embedded page on this page says

11

OK

My First Web Page

My first paragraph.

- **You can skip the window keyword.**
- In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional:

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<script>
alert(5 + 6);
</script>
```

```
</body>
</html>
```

Sample Output

An embedded page on this page says

11

OK

My First Web Page

My first paragraph.

• **Using console.log()**

• For debugging purposes, you can call the console.log() method in the browser to display data.

```
<!DOCTYPE html>
<html>
<body>
<h2>Activate Debugging</h2>
<p>F12 on your keybord will activate debugging.</p>
<p>Then select "Console" in the debugger menu.</p>
<p>Then click Run again.</p>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

Sample Output

Activate Debugging

F12 on your keybord will activate debugging.

Then select "Console" in the debugger menu.

Then click Run again.

- **JavaScript Print**
- JavaScript does not have any print object or print methods.
- You cannot access output devices from JavaScript.
- The only exception is that you can call the window.print() method in the browser to print the content of the current window.

```
<!DOCTYPE html>
<html>
<body>
<h2>The window.print() Method</h2>
<p>Click the button to print the current
page.</p>
<button onclick="window.print()">Print this page</button>
</body>
</html>
```

Sample Output

The window.print() Method

Click the button to print the current page.

Print this page

JavaScript Statements

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Statements</h2>
<p>A <b>JavaScript program</b> is a list of <b>statements</b> to be executed by a
computer.</p>
<p id="demo"></p>
<script>
var x, y, z; // Statement 1
x = 5; // Statement 2
y = 6; // Statement 3
z = x + y; // Statement 4
document.getElementById("demo").innerHTML = "The value of z is " + z + ".";
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

A **JavaScript program** is a list of **statements** to be executed by a computer.

The value of z is 11.

• **JavaScript Programs**

• A **computer program** is a list of "instructions" to be "executed" by a computer.

• In a programming language, these programming instructions are called **statements**.

• A **JavaScript program** is a list of programming **statements**.

• ***In HTML, JavaScript programs are executed by the web browser.***

• **JavaScript Statements**

• JavaScript statements are composed of:

• Values, Operators, Expressions, Keywords, and Comments.

• This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

• **Example**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Statements</h2>
```

```
<p>In HTML, JavaScript statements are executed by the
browser.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "Hello
Dolly.";
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

In HTML, JavaScript statements are executed by the browser.

Hello Dolly.

- Most JavaScript programs contain many JavaScript statements.
- The statements are executed, one by one, in the same order as they are written.
- JavaScript programs (and JavaScript statements) are often called JavaScript code.
- **Semicolons ;**
- Semicolons separate JavaScript statements.
- Add a semicolon at the end of each executable statement:
- var a, b, c;

// Declare 3 variables

```
a = 5;
// Assign the value 5 to a
b = 6;
// Assign the value 6 to b
c = a + b;
// Assign the sum of a and b to c
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Statements</h2>
<p>JavaScript statements are separated by semicolons.</p>
<p id="demo1"></p>
<script>
var a, b, c;
a = 5;
b = 6;
c = a + b;
document.getElementById("demo1").innerHTML = c;
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

JavaScript statements are separated by semicolons.

11

• When separated by semicolons, multiple statements on one line are allowed:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Statements</h2>
<p>Multiple statements on one line is allowed.</p>
<p id="demo1"></p>
<script>
var a, b, c;
a = 5; b = 6; c = a + b;
document.getElementById("demo1").innerHTML = c;
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

Multiple statements on one line is allowed.

11

- **JavaScript White Space**
- JavaScript ignores multiple spaces. You can add white space to your script to make it more readable.
- **The following lines are equivalent:**
- var person = "Hege";

var person="Hege";

- A good practice is to put spaces around operators (= +- * /):
- var x = y + z;
- **JavaScript Line Length and Line Breaks**
- For best readability, programmers often like to avoid code lines longer than 80 characters.
- If a JavaScript statement does not fit on one line, the best place to break it is after an operator:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Statements</h2>
```

```
<p>
The best place to break a code line is after an operator or a
comma.
</p>
<p id="demo"></p>
<scrpt>
document.getElementById("demo").innerHTML =
"Hello Dolly!";
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

The best place to break a code line is after an operator or a comma.

Hello Dolly!

- **JavaScript Code Blocks**
- JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.
- The purpose of code blocks is to define statements to be executed together.
- One place you will find statements grouped together in blocks, is in JavaScript functions:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Statements</h2>
<p>JavaScript code blocks are written between { and }</p>
<button type="button" onclick="myFunction()">Click Me!</button>
<p id="demo1"></p>
<p id="demo2"></p>
<script>
function myFunction() {
document.getElementById("demo1").innerHTML = "Hello Dolly!";
document.getElementById("demo2").innerHTML = "How are you?";
}
</script>
</body>
</html>
```

Sample Output

JavaScript Statements

JavaScript code blocks are written between { and }

Click Me!

JavaScript Statements

JavaScript code blocks are written between { and }

Click Me!

Hello Dolly!

How are you?

- **JavaScript Keywords**
- JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed.
- Visit our Reserved Words reference to view a full list of JavaScript keywords.
- Here is a list of some of the keywords

Keyword	Description
break	Terminates a switch or a loop
continue	Jumps out of a loop and starts at the top
debugger	Stops the execution of JavaScript, and calls (if available) the debugging function
do ... while	Executes a block of statements, and repeats the block, while a condition is true
for	Marks a block of statements to be executed, as long as a condition is true
function	Declares a function
if ... else	Marks a block of statements to be executed, depending on a condition
return	Exits a function
switch	Marks a block of statements to be executed, depending on different cases
try ... catch	Implements error handling to a block of statements
var	Declares a variable

JavaScript Syntax

JavaScript syntax is the set of rules, how JavaScript programs are constructed:

var x, y, z;

// Declare Variables

x = 5; y = 6;

// Assign Values

z = x + y;

// Compute Values

JavaScript Values

The JavaScript syntax defines two types of values:

•Fixed values

•Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

JavaScript Literals

The two most important syntax rules for fixed values are:

1. **Numbers** are written with or without decimals:

10.50

1001

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Numbers</h2>
<p>Number can be written with or without decimals.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 10.50;
</script>
</body>
</html>
```

Sample Output

JavaScript Numbers

Number can be written with or without decimals.

10.5

• 2. **Strings** are text, written within double or single quotes:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Strings</h2>
<p>Strings can be written with double or single quotes.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 'John Doe';
</script>
</body>
</html>
```

Sample Output

JavaScript Strings

Strings can be written with double or single quotes.

John Doe

JavaScript Variables

- In a programming language, **variables** are used to **store** data values.
- JavaScript uses the var keyword to **declare** variables.
- An **equal sign** is used to **assign values** to variables.
- In this example, x is defined as a variable. Then, x is assigned (given) the value 6:
- var x;

x = 6;

- **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p>In this example, x is defined as a variable.
Then, x is assigned the value of 6:</p>
<p id="demo"></p>
<script>
var x;
x = 6;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript Variables

In this example, x is defined as a variable. Then, x is assigned the value of 6:

6

3.2 JavaScript Operators

- JavaScript uses **arithmetic operators** (+ - * /) to **compute** values:

- (5 + 6) * 10
- **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Operators</h2>
<p>JavaScript uses arithmetic operators to compute values (just like algebra).</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = (5 + 6) * 10;
</script>
</body>
</html>
```

Sample Output

JavaScript Operators

JavaScript uses arithmetic operators to compute values (just like algebra).

110

- JavaScript uses an **assignment operator** (=) to **assign** values to variables:
- var x, y;

```
x = 5;
y = 6;
```

```
<!DOCTYPE html>
<html>
<body>
<h2>Assigning JavaScript Values</h2>
<p>In JavaScript the = operator is used to assign values to variables.</p>
<p id="demo"></p>
<script>
var x, y;
x = 5;
y = 6;
document.getElementById("demo").innerHTML = x + y;
</script>
</body>
</html>
```

Sample Output

Assigning JavaScript Values

In JavaScript the = operator is used to assign values to variables.

11

- **JavaScript Expressions**
- An expression is a combination of values, variables, and operators, which computes to a value.
- The computation is called an evaluation.
- For example, 5 * 10 evaluates to 50:
- 5 * 10

- **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Expressions</h2>
<p>Expressions compute to values.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5
* 10;
</script>
</body>
</html>
```

Sample Output

JavaScript Expressions

Expressions compute to values.

50

- Expressions can also contain variable values:
- x * 10

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Expressions</h2>
<p>Expressions compute to values.</p>
<p id="demo"></p>
<script>
var x;
x = 5;
document.getElementById("demo").innerHTML = x * 10;
</script>
</body>
</html>
```

Sample Output

JavaScript Expressions

Expressions compute to values.

50

- The values can be of various types, such as numbers and strings.
- For example, "John" + " " + "Doe", evaluates to "John Doe":
- *"John" + " " + "Doe“*
-

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Expressions</h2>
<p>Expressions compute to values.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "John" + " " + "Doe";
</script>
</body>
</html>
```

Sample Output

JavaScript Expressions

Expressions compute to values.

John Doe

- **JavaScript Keywords**
- JavaScript **keywords** are used to identify actions to be performed.
- The var keyword tells the browser to create variables:
- var x, y;

x = 5 + 6;

y = x * 10;

```
<!DOCTYPE html>
<html>
<body>
<h2>The var Keyword Creates Variables</h2>
<p id="demo"></p>
<script>
var x, y;
x = 5 + 6;
y = x * 10;
document.getElementById("demo").innerHTML = y;
</script>
</body>
</html>
```

Sample Output

The var Keyword Creates Variables

110

JavaScript Comments

• Not all JavaScript statements are "executed".

• Code after double slashes // or between /* and */ is treated as a **comment**.

• Comments are ignored, and will not be executed:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Comments are NOT Executed</h2>
<p id="demo"></p>
<script>
var x;
x = 5;
// x = 6; I will not be executed
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript Comments are NOT Executed

5

3.3 JavaScript Identifiers

• Identifiers are names.

• In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels).

• The rules for legal names are much the same in most programming languages.

• In JavaScript, the first character must be a letter, or an underscore (_), or a dollar sign ($).

• Subsequent characters may be letters, digits, underscores, or dollar signs.

• ***Numbers are not allowed as the first character.***

This way JavaScript can easily distinguish identifiers

from numbers.

• **JavaScript is Case Sensitive**

• All JavaScript identifiers are **case sensitive**.

• The variables lastName and lastname, are two different variables:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript is Case Sensitive</h2>
<p>Try change lastName to lastname.</p>
<p id="demo"></p>
<script>
var lastname, lastName;
lastName = "Doe";
lastname = "Peterson";
document.getElementById("demo").innerHTML = lastName;
</script>
</body>
</html>
```

Sample Output

JavaScript is Case Sensitive

Try change lastName to lastname.

Doe

JavaScript does not interpret **VAR** or **Var** as the keyword **var**.

- **JavaScript and Camel Case**
- Historically, programmers have used different ways of joining multiple words into one variable name:
- **Hyphens:**
- first-name, last-name, master-card, inter-city.
- Hyphens are not allowed in JavaScript. They are reserved for subtractions.
- **Underscore:**
- first_name, last_name, master_card, inter_city.
- **Upper Camel Case (Pascal Case):**
- FirstName, LastName, MasterCard, InterCity.
- **Lower Camel Case:**
- JavaScript programmers tend to use camel case that starts with a lowercase letter:
- firstName, lastName, masterCard, interCity.
- **JavaScript Character Set**
- JavaScript uses the **Unicode** character set.

• Unicode covers (almost) all the characters, punctuations, and symbols in
the world.

3.4 JavaScript Comments

JavaScript comments can be used to explain JavaScript code, and to make it more readable.
JavaScript comments can also be used to prevent execution, when testing alternative
code.

Single Line Comments

Single line comments start with //.

Any text between // and the end of the line will be ignored by JavaScript (will not be executed).

This example uses a single-line comment before each code line:

• **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h1 id="myH"></h1>
<p id="myP"></p>
<script>
// Change heading:
document.getElementById("myH").innerHTML = "JavaScript Comments";
// Change paragraph:
document.getElementById("myP").innerHTML = "My first paragraph.";
</script>
</body>
</html>
```

Sample Output

JavaScript Comments

My first paragraph.

• This example uses a single line comment at the end of each line to explain the code:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Comments</h2>
<p id="demo"></p>
<script>
var x = 5; // Declare x, give it the value of 5
var y = x + 2; // Declare y, give it the value of x + 2
// Write y to demo:
document.getElementById("demo").innerHTML = y;
</script> </body>
</html>
```

Sample Output

JavaScript Comments

7

• **Multi-line Comments**

• Multi-line comments start with /* and end with */.

• Any text between /* and */ will be ignored by JavaScript.

• This example uses a multi-line comment (a comment block) to explain the code:

```
<!DOCTYPE html>
<html> <body>
<h1 id="myH"></h1>
<p id="myP"></p>
<script>
/*
The code below will change
the heading with id = "myH"
and the paragraph with id = "myP"
*/
document.getElementById("myH").innerHTML = "JavaScript Comments";
document.getElementById("myP").innerHTML = "My first paragraph.";
</script> </body> </html>
```

Sample Output

JavaScript Comments

My first paragraph.

It is most common to use single line comments. Block comments are often used for formal documentation.

- **Using Comments to Prevent Execution**
- Using comments to prevent execution of code is suitable for code testing.
- Adding // in front of a code line changes the code lines from an executable line to a comment.
- This example uses // to prevent execution of one of the code lines:<!DOCTYPE html>

```
<html>
<body>
<h2>JavaScript Comments</h2>
<h1 id="myH"></h1>
<p id="myP"></p>
<script>
//document.getElementById("myH").innerHTML = "My
First Page";
document.getElementById("myP").innerHTML = "My first
paragraph.";
</script>
<p>The line starting with // is not executed.</p>
</body>
</html>
```

Sample Output

JavaScript Comments

My first paragraph.

The line starting with // is not executed.

• This example uses a comment block to prevent execution of multiple lines:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Comments</h2>
<h1 id="myH"></h1>
<p id="myP"></p>
<script>
/*
document.getElementById("myH").innerHTML = "Welcome to my
Homepage";
document.getElementById("myP").innerHTML = "This is my first
paragraph.";
*/
document.getElementById("myP").innerHTML = "The comment-block
is not executed.";
</script> </body> </html>
```

Sample Output

JavaScript Comments

The comment-block is not executed.

3.5 JavaScript Variables

JavaScript variables are containers for storing data values.

In this example, x, y, and z, are variables:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Variables</h2>
<p>In this example, x, y, and z are variables.</p>
<p id="demo"></p>
<script>
var x = 5;
var y = 6;
var z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script> </body> </html>
```

Sample Output

JavaScript Variables

In this example, x, y, and z are variables.

The value of z is: 11

From the example above, you can expect:

x stores the value 5

y stores the value 6

z stores the value 11

- **Using let and const (2015)**
- Before 2015, using the var keyword was the only way to declare a JavaScript variable.

• The 2015 version of JavaScript (ES6 - ECMAScript 2015) allows the use of the ***const*** keyword to define a variable that cannot be reassigned, and the ***let*** keyword to define a variable with restricted scope.

• Because it is a little complicated to describe the difference between these keywords, and because they are not supported in older browsers, the first part of this tutorial will most often use var.

Safari 10 and Edge 14 were the first browsers to fully support ES6:

Chrome 58	Edge 14	Firefox 54	Safari 10	Opera 55
Jan 2017	Aug 2016	Mar 2017	Jul 2016	Aug 2018

• **Much Like Algebra**

• In this example, price1, price2, and total, are variables:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p id="demo"></p>
<script>
var price1 = 5;
```

```
var price2 = 6;
var total = price1 + price2;
document.getElementById("demo").innerHTML =
"The total is: " + total;
</script> </body> </html>
```

Sample Output

JavaScript Variables

The total is: 11

- In programming, just like in algebra, we use variables (like price1) to hold values.
- In programming, just like in algebra, we use variables in expressions (total = price1 + price2).
- From the example above, you can calculate the total to be 11.
- JavaScript variables are containers for storing data values.

- **JavaScript Identifiers**
- All JavaScript **variables** must be **identified** with **unique names**.
- These unique names are called **identifiers**.
- Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

• The general rules for constructing names for variables (unique identifiers) are:

• Names can contain letters, digits, underscores, and dollar signs.

• Names must begin with a letter

• Names can also begin with $ and _ (but we will not use it in this tutorial)

• Names are case sensitive (y and Y are different variables)

• Reserved words (like JavaScript keywords) cannot be used as names

• JavaScript identifiers are case-sensitive.

• **The Assignment Operator**

• In JavaScript, the equal sign (=) is an "assignment" operator, not an "equal to" operator.

• This is different from algebra. The following does not make sense in algebra:

• x = x + 5

• In JavaScript, however, it makes perfect sense: it assigns the value of x + 5 to x.

• (It calculates the value of x + 5 and puts the result into x. The value of x is incremented by 5.)

• The "equal to" operator is written like == in JavaScript.

• **JavaScript Data Types**

• JavaScript variables can hold numbers like 100 and text values like "John Doe".

• In programming, text values are called text strings.

• JavaScript can handle many types of data, but for now, just think of numbers and strings.

• Strings are written inside double or single quotes. Numbers are written without quotes.

• If you put a number in quotes, it will be treated as a text string.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p>Strings are written with quotes.</p>
<p>Numbers are written without quotes.</p>
<p id="demo"></p>
```

```
<script>
var pi = 3.14;
var person = "John Doe";
var answer = 'Yes I am!';
document.getElementById("demo").innerHTML =pi + "<br>" + person + "<br>" + answer;
</script> </body> </html>
```

Sample Output

JavaScript Variables

Strings are written with quotes.

Numbers are written without quotes.

3.14
John Doe
Yes I am!

- **Declaring (Creating) JavaScript Variables**
- Creating a variable in JavaScript is called "declaring" a variable.
- You declare a JavaScript variable with the var keyword:

- var carName;
- After the declaration, the variable has no value (technically it has the value of undefined).
- To **assign** a value to the variable, use the equal sign:
- carName = "Volvo";
- You can also assign a value to the variable when you declare it:
- var carName = "Volvo";• In the example below, we create a variable called carName and assign the value "Volvo" to it.

• Then we "output" the value inside an HTML paragraph with id="demo":

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Variables</h2>
<p>Create a variable, assign a value to it, and display
it:</p>
<p id="demo"></p>
<script>
var carName = "Volvo";
document.getElementById("demo").innerHTML =
carName;
</script> </body> </html>
```

Sample Output

JavaScript Variables

Create a variable, assign a value to it, and display it:

Volvo

• **One Statement, Many Variables**

• You can declare many variables in one statement.

• Start the statement with var and separate the variables by **comma**:

```
<!DOCTYPE html>
<html> <body>
```

```
<h2>JavaScript Variables</h2>
<p>You can declare many variables in one statement.</p>
<p id="demo"></p>
<script>
var person = "John Doe", carName = "Volvo", price = 200;
document.getElementById("demo").innerHTML =
carName;
</script> </body> </html>
```

Sample Output

JavaScript Variables

You can declare many variables in one statement.

Volvo

• **A declaration can span multiple lines:**

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Variables</h2>
<p>You can declare many variables in one statement.</p>
<p id="demo"></p>
<script>
var person = "John Doe",
```

```
carName = "Volvo",
price = 200;
document.getElementById("demo").innerHTML = carName;
</script> </body> </html>
```

Sample Output

JavaScript Variables

You can declare many variables in one statement.

Volvo

• **Value = undefined**

• In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.

• A variable declared without a value will have the value undefined.

• The variable carName will have the value undefined after the execution of this statement:

• **Example:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
```

```
<p>A variable declared without a value will have the
value undefined.</p>
<p id="demo"></p>
<script>
var carName;
document.getElementById("demo").innerHTML =
carName;
</script>
</body>
</html>
```

Sample Output

JavaScript Variables

A variable declared without a value will have the value undefined.

undefined

- **Re-Declaring JavaScript Variables**
- If you re-declare a JavaScript variable, it will not lose its value.
- The variable carName will still have the value "Volvo" after the execution of these statements:
- **Example:**

```
<!DOCTYPE html>
```

```
<html> <body>
<h2>JavaScript Variables</h2>
<p>If you re-declare a JavaScript variable, it will not lose its value.</p>
<p id="demo"></p>
<script>
var carName = "Volvo";
var carName;
document.getElementById("demo").innerHTML =carName;
</script> </body> </html>
```

Sample Output

JavaScript Variables

If you re-declare a JavaScript variable, it will not lose its value.

Volvo

- **JavaScript Arithmetic**
- As with algebra, you can do arithmetic with JavaScript variables, using operators like = and +:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Variables</h2>
```

```
<p>The result of adding 5 + 2 + 3:</p>
<p id="demo"></p>
<script>
var x = 5 + 2 + 3;
document.getElementById("demo").innerHTML = x;
</script> </body> </html>
```

Sample Output

JavaScript Variables

The result of adding 5 + 2 + 3:

10

• **You can also add strings, but strings will be concatenated:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p>The result of adding "John" + " " + "Doe":</p>
<p id="demo"></p>
<script>
var x = "John" + " " + "Doe";
document.getElementById("demo").innerHTML = x;
```

```
</script>
</body>
</html>
```

Sample Output

JavaScript Variables

The result of adding "John" + " " + "Doe":

John Doe

- **Also try this:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p>The result of adding "5" + 2 + 3:</p>
<p id="demo"></p>
<script>
x = "5" + 2 + 3;
document.getElementById("demo").innerHTML = x;
</script>
</body>
```

```
</html>
```

Sample Output

JavaScript Variables

The result of adding "5" + 2 + 3:

523

If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.

- **Now try this:**

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Variables</h2>
<p>The result of adding 2 + 3 + "5":</p>
<p id="demo"></p>
<script>
var x = 2 + 3 + "5";
document.getElementById("demo").innerHTML = x;
</script> </body> </html>
```

Sample Output

JavaScript Variables

The result of adding 2 + 3 + "5":

55

- **JavaScript Dollar Sign $**
- Remember that JavaScript identifiers (names) must begin with:
- A letter (A-Z or a-z)
- A dollar sign ($)
- Or an underscore (_)
- Since JavaScript treats a dollar sign as a letter, identifiers containing $ are valid variable names:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript $</h2>
<p>The dollar sign is treated as a letter in JavaScript
names.</p>
<p id="demo"></p>
```

```
<script>
var $ = 2;
var $myMoney = 5;
document.getElementById("demo").innerHTML = $ +
$myMoney;</script>
</body>
</html>
```

Sample Output

JavaScript $

The dollar sign is treated as a letter in JavaScript names.

7

- Using the dollar sign is not very common in JavaScript, but professional programmers often use it as an alias for the main function in a JavaScript library.
- In the JavaScript library jQuery, for instance, the main function $ is used to select HTML elements. In jQuery $("p"); means "select all p elements".

- **JavaScript Underscore (_)**
- Since JavaScript treats underscore as a letter, identifiers containing _ are valid variable names:

```
<!DOCTYPE html>
```

```
<html> <body>
<h2>JavaScript $</h2>
<p>The underscore is treated as a letter in JavaScript
names.</p>
<p id="demo"></p>
<script>
var _x = 2;
var _100 = 5;
document.getElementById("demo").innerHTML = _x +
_100;
</script> </body> </html>
```

Sample Output

JavaScript $

The underscore is treated as a letter in JavaScript names.

7

Using the underscore is not very common in JavaScript, but a convention among professional programmers is to use it as an alias for "private (hidden)" variables.

3.6 JavaScript Operators

Example

Assign values to variables and add them together:

```
var x = 5;
// assign the value 5 to x
var y = 2;
// assign the value 2 to y
var z = x + y;
```

```
// assign the value 7 to z (x + y)

<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Operators</h2>
<p>x = 5, y = 2, calculate z = x + y, and display z:</p>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x + y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

JavaScript Operators

x = 5, y = 2, calculate z = x + y, and display z:

7

• The **assignment** operator (=) assigns a value to a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The = Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The = Operator

10

• The **addition** operator (+) adds numbers:

```
<!DOCTYPE html>
<html>
<body>
<h2>The + Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x + y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The + Operator

7

• The **multiplication** operator (*) multiplies numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The * Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x * y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The * Operator

10

3.7 JavaScript Arithmetic Operators

Arithmetic operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

• The **addition assignment** operator (+=) adds a value to a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The += Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x += 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The += Operator

15

- **JavaScript String Operators**
- The + operator can also be used to add (concatenate) strings.

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Operators</h2>
<p>The + operator concatenates (adds) strings.</p>
<p id="demo"></p>
<script>
var txt1 = "John";
var txt2 = "Doe";
document.getElementById("demo").innerHTML = txt1 + " " + txt2;
</script> </body> </html>
```

Sample Output

JavaScript Operators

The + operator concatenates (adds) strings.

John Doe

• The += assignment operator can also be used to add (concatenate) strings:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Operators</h2>
<p>The assignment operator += can concatenate strings.</p>
<p id="demo"></p>
<script>
txt1 = "What a very ";
txt1 += "nice day";
document.getElementById("demo").innerHTML = txt1;
</script>
</body>
</html>
```

Sample Output

JavaScript Operators

The assignment operator += can concatenate strings.

What a very nice day

• **Adding Strings and Numbers**

• Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Operators</h2>
<p>Adding a number and a string, returns a string.</p>
<p id="demo"></p>
<script>
var x = 5 + 5;
var y = "5" + 5;
var zdocument.getElementById("demo").innerHTML =
= "Hello" + 5;
x + "<br>" + y + "<br>" + z;
</script>
</body>
</html>
```

Sample Output

JavaScript Operators

Adding a number and a string, returns a string.

10
55
Hello5

JavaScript Comparison Operators

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	Zero fill left shift	5 << 1	0101 << 1	1010	10
>>	Signed right shift	5 >> 1	0101 >> 1	0010	2
>>>	Zero fill right shift	5 >>> 1	0101 >>> 1	0010	2

JavaScript Arithmetic Operators

Arithmetic operators perform arithmetic on numbers (literals or variables).

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Remainder)
++	Increment
--	Decrement

- **Arithmetic Operations**
- A typical arithmetic operation operates on two numbers.
- The two numbers can be literals:

```
<!DOCTYPE html>
<html> <body>
<p>A typical arithmetic operation takes two
numbers and produces a new number.</p>
<p id="demo"></p>
<script>
var x = 100 + 50;
document.getElementById("demo").innerHTML = x;
</script> </body> </html>
```

Sample Output

A typical arithmetic operation takes two numbers and produces a new number.

150

• **The two numbers can be variables:**

```
<!DOCTYPE html>
<html>
<body>
<p>A typical arithmetic operation takes two numbers (or variables) and produces a new
number.</p>
<p id="demo"></p>
<script>
var a = 100;
var b = 50;
var x = a + b;
document.getElementById("demo").innerHTML = x;
</script>
```

```
</body>
</html>
```

Sample Output

A typical arithmetic operation takes two numbers (or variables) and produces a new number.

150

• **The two numbers can be expressions:**

```
<!DOCTYPE html>
<html>
<body>
<p>A typical arithmetic operation takes two numbers (or expressions) and produces a new
number.</p>
<p id="demo"></p>
<script>
var a = 3;
var x = (100 + 50) * a;
document.getElementById("demo").innerHTML = x;
</script>
</body>
```

</html>

Sample Output

A typical arithmetic operation takes two numbers (or expressions) and produces a new number.

450

Operators and Operands

The numbers (in an arithmetic operation) are called **operands.**

The operation (to be performed between the two operands) is defined by an **operator.**

Operand	Operator	Operand
100	+	50

- **Adding**
- The **addition** operator (+) adds numbers:

```
<!DOCTYPE html>
<html>
<body>
<h2>The + Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x + y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The + Operator

7

- **Subtracting**
- The **subtraction** operator (-) subtracts numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The - Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x - y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The - Operator

3

• **Multiplying**

• The **multiplication** operator (*) multiplies numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The * Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x * y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The * Operator

10

- **Dividing**
- The **division** operator (/) divides numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The / Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x / y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The / Operator

2.5

• **Remainder**

• The **modulus** operator (%) returns the division remainder.

```
<!DOCTYPE html>
<html>
<body>
<h2>The % Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
var y = 2;
var z = x % y;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The % Operator

1

In arithmetic, the division of two integers produces a **quotient** and a **remainder**.

In mathematics, the result of a **modulo operation** is the **remainder** of an arithmetic division.

- **Incrementing**
- The **increment** operator (++) increments numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The ++ Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
x++;
var z = x;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The ++ Operator

6

- **Decrementing**
- The **decrement** operator (--) decrements numbers.

```
<!DOCTYPE html>
<html>
<body>
<h2>The -- Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
x--;
var z = x;
document.getElementById("demo").innerHTML = z;
</script>
</body>
</html>
```

Sample Output

The -- Operator

4

- **Exponentiation**
- The **exponentiation** operator (**) raises the first operand to the power of the second operand.

```
<!DOCTYPE html>
<html>
<body>
<h2>The ** Operator</h2>
<p id="demo"></p>
<script>
var x = 5;
document.getElementById("demo").innerHTML = x** 2;
</script>
</body>
</html>
```

Sample Output

The ** Operator

25

• **x ** y produces the same result as Math.pow(x,y):**

```
<!DOCTYPE html>
<html>
<body>
<h2>Math.pow()</h2>
<p id="demo"></p>
<script>
var x = 5;
document.getElementById("demo").innerHTML =
Math.pow(x,2);
</script>
</body>
</html>
```

Sample Output

Math.pow()

25

• **Operator Precedence**

• Operator precedence describes the order in which operations are performed in an arithmetic expression.

```
<!DOCTYPE html>
<html>
<body>
<p>Multiplication has precedence over addition.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 100 +
50 * 3;
</script>
</body>
</html>
```

Sample Output

Multiplication has precedence over addition.

250

JavaScript Operator Precedence Values

Value	Operator	Description	Example
20	( )	Expression grouping	(3 + 4)
19	.	Member	person.name
19	[]	Member	person["name"]
19	()	Function call	myFunction()
19	new	Create	new Date()
17	++	Postfix Increment	i++
17	--	Postfix Decrement	i--
16	++	Prefix Increment	++i
16	--	Prefix Decrement	--i
16	!	Logical not	!(x==y)
16	typeof	Type	typeof x
15	**	Exponentiation (ES2016)	10 ** 2
14	*	Multiplication	10 * 5
14	/	Division	10 / 5
14	%	Division Remainder	10 % 5

13	+	Addition	10 + 5
13	-	Subtraction	10 - 5
12	<<	Shift left	x << 2
12	>>	Shift right	x >> 2
12	>>>	Shift right (unsigned)	x >>> 2
11	<	Less than	x < y
11	<=	Less than or equal	x <= y
11	>	Greater than	x > y
11	>=	Greater than or equal	x >= y
11	in	Property in Object	"PI" in Math
11	instanceof	Instance of Object	instanceof Array
10	==	Equal	x == y
10	===	Strict equal	x === y
10	!=	Unequal	x != y
10	!==	Strict unequal	x !== y
9	&	Bitwise AND	x & y
8	^	Bitwise XOR	x ^ y

7	|	Bitwise OR	x | y
6	&&	Logical AND	x && y
5	||	Logical OR	x || y
4	? :	Condition	? "Yes" : "No"
3	+=	Assignment	x += y
3	/=	Assignment	x /= y
3	-=	Assignment	x -= y
3	*=	Assignment	x *= y
3	%=	Assignment	x %= y
3	<<=	Assignment	x <<= y
3	>>=	Assignment	x >>= y
3	>>>=	Assignment	x >>>= y
3	&=	Assignment	x &= y
3	^=	Assignment	x ^= y
3	|=	Assignment	x |= y
2	yield	Pause Function	yield x
1	,	Comma	5 , 6

Expressions in parentheses are fully computed before the value is used in the rest of the expression.

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
<<=	x <<= y	x = x << y
>>=	x >>= y	x = x >> y
>>>=	x >>>= y	x = x >>> y
&=	x &= y	x = x & y
^=	x ^= y	x = x ^ y
|=	x |= y	x = x | y
**=	x **= y	x = x ** y

The `**=` operator is an experimental part of the ECMAScript 2016 proposal (ES7). It is not stable across browsers. Do not use it.

Assignment Examples

The = assignment operator assigns a value to a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The = Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

• The += assignment operator adds a value to a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The += Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x += 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The += Operator

15

• The -= assignment operator subtracts a value from a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The -= Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x -= 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The -= Operator

5

• The *= assignment operator multiplies a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The *= Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x *= 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The *= Operator

50

• The /= assignment divides a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The /= Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x /= 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The /= Operator

2

• The %= assignment operator assigns a remainder to a variable.

```
<!DOCTYPE html>
<html>
<body>
<h2>The %= Operator</h2>
<p id="demo"></p>
<script>
var x = 10;
x %= 5;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

The %= Operator

0

3.8 JavaScript Data Types

JavaScript variables can hold many **data types**: numbers, strings, objects and more:

• var length = 16;

// Number

• var lastName = "Johnson";

// String

•var x = {firstName:"John", lastName:"Doe"}; //

Object

The Concept of Data Types

In programming, data types is an important concept. To be able to operate on variables, it is important to know something about the type. Without data types, a computer cannot safely solve this:

var x = 16 + "Volvo";

• Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

• JavaScript will treat the example above as:

• var x = "16" + "Volvo";

• When adding a number and a string, JavaScript will treat the number as a string.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>When adding a number and a string, JavaScript will
treat the number as a string.</p>
<p id="demo"></p>
<script>
var x = 16 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>
</body>
```

```
</html>
```

Sample Output

JavaScript

When adding a number and a string, JavaScript will treat the number as a string.

16Volvo

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>When adding a string and a number, JavaScript will treat the number as a string.</p>
<p id="demo"></p>
<script>
var x = "Volvo" + 16;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript

When adding a string and a number, JavaScript will treat the number as a string.

Volvo16

• JavaScript evaluates expressions from left to right. Different sequences can produce different results:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>JavaScript evaluates expressions from left to right. Different sequences can produce different
results:</p>
<p id="demo"></p>
<script>
var x = 16 + 4 + "Volvo";
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

20Volvo

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>JavaScript evaluates expressions from left to right. Different sequences can produce different
results:</p>
<p id="demo"></p>
<script>
var x = "Volvo" + 16 + 4;
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

Volvo164

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".

In the second example, since the first operand is a string, all operands are treated as strings.

• **JavaScript Types are Dynamic**

• JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

• **Example**

• var x;

// Now x is undefined

x = 5;

// Now x is a Number

x = "John";

// Now x is a String<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Data Types</h2>

<p>JavaScript has dynamic types. This means that the same variable can be used to hold different data types:</p>

<p id="demo"></p>

<script>

```
var x; // Now x is undefined
x = 5; // Now x is a Number
x = "John"; // Now x is a String
document.getElementById("demo").innerHTML = x;
</script>
</body>
</html>
```

Sample Output

JavaScript Data Types

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

John

- **JavaScript Strings**
- A string (or a text string) is a series of characters like "John Doe".
- Strings are written with quotes. You can use single or double quotes:
- **Example**
- var carName1 = "Volvo XC60"; // Using double quotes

```
var carName2 = 'Volvo XC60'; // Using single quotes<!DOCTYPE html>
<html>
<body>
```

```
<h2>JavaScript Strings</h2>
<p>Strings are written with quotes. You can use single or double quotes:</p>
<p id="demo"></p>
<script>
var carName1 = "Volvo XC60";
var carName2 = 'Volvo XC60';
document.getElementById("demo").innerHTML = carName1 + "<br>" +
carName2;
</script>
</body>
</html>
```

Sample Output

JavaScript Strings

Strings are written with quotes. You can use single or double quotes:

Volvo XC60
Volvo XC60

• You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

• **Example**

1) var answer1 = "It's alright";

// Single

quote inside double quotes

2) var answer2 = "He is called 'Johnny'"; //

Single quotes inside double quotes

3)var answer3 = 'He is called "Johnny"'; //

Double quotes inside single quotes<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Strings</h2>

<p>You can use quotes inside a string, as long as they don't match the

quotes surrounding the string:</p>

<p id="demo"></p>

<script>

var answer1 = "It's alright";

var answer2 = "He is called 'Johnny'";

var answer3 = 'He is called "Johnny"';

document.getElementById("demo").innerHTML =

answer1 + "<br>" +

answer2 + "<br>" +

answer3;

</script>

</body>

</html>

Sample Output

JavaScript Strings

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

It's alright
He is called 'Johnny'
He is called "Johnny"

- **JavaScript Numbers**
- JavaScript has only one type of numbers.
- Numbers can be written with, or without decimals:
- **Example**
- var x1 = 34.00;

// Written with decimals

var x2 = 34;

// Written without decimals

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Numbers</h2>
<p>Numbers can be written with, or without decimals:</p>
<p id="demo"></p>
<script>
var x1 = 34.00;
var x2 = 34;
var x3 = 3.14;
document.getElementById("demo").innerHTML = x1 + "<br>" + x2 + "<br>" + x3;
</script>
</body>
</html>
```

Sample Output

JavaScript Numbers

Numbers can be written with, or without decimals:

34
34
3.14

• Extra large or extra small numbers can be written with scientific (exponential) notation:

• **Example**

```
• var y = 123e5;
// 12300000
var z = 123e-5;
// 0.00123<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Numbers</h2>
<p>Extra large or extra small numbers can be written with scientific (exponential) notation:</p>
<p id="demo"></p>
<script>
var y = 123e5;
var z = 123e-5;
document.getElementById("demo").innerHTML =
y + "<br>" + z;
</script>
</body>
</html>
```

Sample Output

JavaScript Numbers

Extra large or extra small numbers can be written with scientific (exponential) notation:

12300000
0.00123

- **JavaScript Booleans**
- Booleans can only have two values: true or false.
- **Example**
- var x = 5;

```
var y = 5;
var z = 6;
(x == y)
// Returns true
(x == z)
// Returns false<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Booleans</h2>
<p>Booleans can have two values: true or false:</p>
<p id="demo"></p>
<script>
var x = 5;
var y = 5;
var z = 6;
document.getElementById("demo").innerHTML =
(x == y) + "<br>" + (x == z);
</script>
</body>
</html>
```

Sample Output

JavaScript Booleans

Booleans can have two values: true or false:

true
false

JavaScript Arrays

• JavaScript arrays are written with square brackets.

• Array items are separated by commas.

• The following code declares (creates) an array called cars, containing three items (car names):

• Example

• var cars = ["Saab", "Volvo", "BMW"];

• Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

• JavaScript Objects

• JavaScript objects are written with curly braces {}.

• Object properties are written as name:value pairs, separated by commas.

• Example

• var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

• The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

• The typeof Operator

• You can use the JavaScript typeof operator to find the type of a JavaScript variable.

• The typeof operator returns the type of a variable or an expression:

• Example

• typeof "" // Returns "string"

typeof "John" // Returns "string"

typeof "John Doe" // Returns "string"

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript typeof</h2>
<p>The typeof operator returns the type of a variable or an
expression.</p>
<p id="demo"></p>
<script>
```

```
document.getElementById("demo").innerHTML =
typeof "" + "<br>" +
typeof "John" + "<br>" +
typeof "John Doe";
</script>
</body>
</html>
```

Sample Output

JavaScript typeof

The typeof operator returns the type of a variable or an expression.

string
string
string

- **Example**
- typeof 0 // Returns "number"

typeof 314 // Returns "number"

typeof 3.14 // Returns "number"

typeof (3) // Returns "number"

typeof (3 + 4) // Returns "number"

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript typeof</h2>
<p>The typeof operator returns the type of a variable or an
expression.</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML =
typeof 0 + "<br>" +
typeof 314 + "<br>" +
typeof 3.14 + "<br>" +
typeof (3) + "<br>" +
typeof (3 + 4);
</script>
</body>
</html>
```

Sample Output

JavaScript typeof

The typeof operator returns the type of a variable or an expression.

number
number
number
number
number

• **Undefined**

• In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

• **Example**

• var car; // Value is undefined, type is undefined

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>The value (and the data type) of a variable with no value is <b>undefined</b>.</p>
<p id="demo"></p>
<script>
var car;
document.getElementById("demo").innerHTML = car + "<br>" + typeof car;
</script>
</body>
</html>
```

Sample Output

JavaScript

The value (and the data type) of a variable with no value is **undefined**.

undefined
undefined

• Any variable can be emptied, by setting the value to undefined. The type will also be undefined.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>Variables can be emptied if you set the value to <b>undefined</b>.</p>
<p id="demo"></p>
<script>
var car = "Volvo";
car = undefined;
document.getElementById("demo").innerHTML = car + "<br>" + typeof car;
</script>
</body>
</html>
```

Sample Output

JavaScript

Variables can be emptied if you set the value to **undefined**.

undefined
undefined

- **Empty Values**
- An empty value has nothing to do with undefined.
- An empty string has both a legal value and a type.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>An empty string has both a legal value and a type:</p>
<p id="demo"></p>
<script>
var car = "";
document.getElementById("demo").innerHTML = "The value is: " +
car + "<br>" + "The type is: " + typeof car;
</script>
</body>
</html>
```

Sample Output

JavaScript

An empty string has both a legal value and a type:

The value is:
The type is: string

- **Null**
- In JavaScript null is "nothing". It is supposed to be something that doesn't exist.
- Unfortunately, in JavaScript, the data type of null is an object.
- You can consider it a bug in JavaScript that typeof null is an object. It should be null.
- You can empty an object by setting it to null:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>Objects can be emptied by setting the value to <b>null</b>.</p>
<p id="demo"></p>
<script>
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = null;
document.getElementById("demo").innerHTML = typeof person;
</script>
</body>
</html>
```

Sample Output

JavaScript

Objects can be emptied by setting the value to **null**.

object

• You can also empty an object by setting it to undefined:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>Objects can be emptied by setting the value to <b>undefined</b>.</p>
<p id="demo"></p>
<script>
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
person = undefined;
document.getElementById("demo").innerHTML = person;
</script>
</body>
</html>
```

Sample Output

JavaScript

Objects can be emptied by setting the value to **undefined**.

undefined

• **Difference Between Undefined and Null**

• undefined and null are equal in value but different in type:

• typeof undefined // undefined

typeof null // object

null === undefined // false

null == undefined // true

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript</h2>
<p>Undefined and null are equal in value but different in type:</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = typeof undefined + "<br>" +
typeof null + "<br><br>" + (null === undefined) + "<br>" + (null == undefined);
</script>
</body>
</html>
```

Sample Output

JavaScript

Undefined and null are equal in value but different in type:

undefined
object

false
true

• **Primitive Data**

• A primitive data value is a single simple data value with no additional properties and methods.

• The typeof operator can return one of these primitive types:

• string

• number

• boolean

• Undefined

• **Example**

• typeof "John" // Returns "string"

typeof 3.14 // Returns "number"

typeof true // Returns "boolean"

typeof false // Returns "boolean"

typeof x // Returns "undefined" (if x has no value)

• **Complex Data**

• The typeof operator can return one of two complex types:

• function

• object

• The typeof operator returns "object" for objects, arrays, and null.

• The typeof operator does not return "object" for functions.

• **Example**

• typeof {name:'John', age:34} // Returns "object"

typeof [1,2,3,4] // Returns "object" (not "array", see note below)

typeof null // Returns "object"

typeof function myFunc(){} // Returns "function"

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

Example

```
function myFunction(p1, p2) {
return p1 * p2; // The function returns the product of p1 and p2 }
```

3.9 JavaScript Function Syntax

• A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses**()**.

• Function names can contain letters, digits, underscores,
and dollar signs (same rules as variables).

• The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

• The code to be executed, by the function, is placed inside curly brackets: **{}**

• function *name*(*parameter1, parameter2, parameter3*)

• {
// *code to be executed*
}

• Function **parameters** are listed inside the parentheses () in the function definition.

• Function **arguments** are the **values** received by the function when it is invoked.

• Inside the function, the arguments (the parameters) behave as local variables.

• A Function is much the same as a Procedure or a Subroutine, in other programming languages.

• **Function Invocation**

• The code inside the function will execute when "something" **invokes** (calls) the function:

• When an event occurs (when a user clicks a button)

• When it is invoked (called) from JavaScript code

• Automatically (self invoked)

• **Function Return**

• When JavaScript reaches a return statement, the function will stop executing.

• If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

• Functions often compute a **return value**. The return value is "returned" back to the "caller":

• **Example**

• Calculate the product of two numbers, and return the result:

• var x = myFunction(4, 3); // Function is called, return value will end up in x

```
function myFunction(a, b) {
return a * b;
// Function returns the product of a and b
}
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Functions</h2>
<p>This example calls a function which performs a calculation and returns the result:</p>
<p id="demo"></p>
<script>
var x = myFunction(4, 3);
document.getElementById("demo").innerHTML = x;
function myFunction(a, b) {
return a * b;
}
</script>
</body>
</html>
```

Sample Output

JavaScript Functions

This example calls a function which performs a calculation and returns the result:

12

- **Why Functions?**
- You can reuse code: Define the code once, and use it many times.
- You can use the same code many times with different arguments, to produce different results.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Functions</h2>
<p>This example calls a function to convert from Fahrenheit to Celsius:</p>
<p id="demo"></p>
<script>
function toCelsius(f) {
return (5/9) * (f-32);
}
document.getElementById("demo").innerHTML =
toCelsius(77);
</script>
</body>
</html>
```

Sample Output

JavaScript Functions

This example calls a function to convert from Fahrenheit to Celsius:

25

• **The ( ) Operator Invokes the Function**

• Using the example above, toCelsius refers to the function object, and toCelsius() refers to the function result.

• Accessing a function without () will return the function object instead of the function result.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Functions</h2>
<p>Accessing a function without () will return the function definition instead of the function result:</p>
<p id="demo"></p>
<script>
function toCelsius(f) {
return (5/9) * (f-32);
}
document.getElementById("demo").innerHTML = toCelsius;
</script>
</body>
</html>
```

Sample Output

JavaScript Functions

Accessing a function without () will return the function definition instead of the function result:

function toCelsius(f) { return (5/9) * (f-32); }

• **Functions Used as Variable Values**

• Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

• **Example**

• Instead of using a variable to store the return value of a function:

• var x = toCelsius(77);
var text = "The temperature is " + x + " Celsius";

• You can use the function directly, as a variable value:

• var text = "The temperature is " + toCelsius(77) + " Celsius";

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Functions</h2>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = "The temperature is " + toCelsius(77) + "
Celsius";
function toCelsius(fahrenheit) {
return (5/9) * (fahrenheit-32);
}
</script>
</body>
</html>
```

Sample Output

The temperature is 25 Celsius

• **Local Variables**

• Variables declared within a JavaScript function, become **LOCAL** to the function.

• Local variables can only be accessed from within the function.

• Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

• Local variables are created when a function starts, and deleted when the function is completed.

• **Example**

• // code here can NOT use carName function myFunction() {

var carName = "Volvo";

// code here CAN use carName

}

// code here can NOT use carName

<!DOCTYPE html>

<html>

```
<body>
<h2>JavaScript Functions</h2>
<p>Outside myFunction() carName is undefined.</p>
<p id="demo1"></p>
<p id="demo2"></p>
<script>
myFunction();
function myFunction() {
var carName = "Volvo";
document.getElementById("demo1").innerHTML =
typeof carName + " " + carName;
}
document.getElementById("demo2").innerHTML =
typeof carName;
</script> </body> </html>
```

Sample Output

JavaScript Functions

Outside myFunction() carName is undefined.

string Volvo

undefined

3.10 JavaScript Arrays

JavaScript arrays are used to store multiple values in a single variable.

Example

var cars = ["Saab", "Volvo", "BMW"];

What is an Array? An array is a special variable, which can hold more
than one value at a time.

If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this:

var car1 = "Saab";

var car2 = "Volvo";

var car3 = "BMW";

- However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300?

- **The solution is an array!**
- An array can hold many values under a single name, and you can access the values by referring to an index number.
- **Creating an Array**
- Using an array literal is the easiest way to create a JavaScript Array.
- **Syntax:**
- var *array_name* = [*item1*, *item2*, ...];
- **Example**
- var cars = ["Saab", "Volvo", "BMW"];<!DOCTYPE html>

```
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
```

```
</body>
</html>
```

Sample Output

JavaScript Arrays

Saab,Volvo,BMW

• **Spaces and line breaks are not important. A declaration can span multiple lines:**

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = [
"Saab",
"Volvo",
"BMW"
];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
```

```
</html>
```

Sample Output

JavaScript Arrays

Saab,Volvo,BMW

- **Using the JavaScript Keyword new**
- The following example also creates an Array, and assigns values to it:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = new Array("Saab", "Volvo", "BMW");
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Saab,Volvo,BMW

- **Access the Elements of an Array**
- You access an array element by referring to the **index number**.
- This statement accesses the value of the first element in cars:
- var name = cars[0];

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars[0];
</script>
</body>
```

```
</html>
```

Sample Output

JavaScript Arrays

JavaScript array elements are accessed using numeric indexes (starting from 0).

Saab

- **Changing an Array Element**
- This statement changes the value of the first element in cars:
- cars[0] = "Opel";

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accessed using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Opel";
document.getElementById("demo").innerHTML = cars;
```

```
</script> </body> </html>
```

Sample Output

JavaScript Arrays

JavaScript array elements are accessed using numeric indexes (starting from 0).

Opel,Volvo,BMW

• **Access the Full Array**

• With JavaScript, the full array can be accessed by referring to the array name:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Saab,Volvo,BMW

- **Arrays are Objects**
- Arrays are a special type of objects. The typeof operator in JavaScript returns "object" for arrays.
- But, JavaScript arrays are best described as arrays.
- Arrays use **numbers**to access its "elements". In this example, person[0] returns John:
- **Array:**
- var person = ["John", "Doe", 46];<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Arrays</h2>

<p>Arrays use numbers to access its elements.</p>

<p id="demo"></p>

<script>

var person = ["John", "Doe", 46];

document.getElementById("demo").innerHTML = person[0];

```
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Arrays use numbers to access its elements.

John

• Objects use **names** to access its "members". In this example, person.firstName returns John:

• **Object:**

• var person = {firstName:"John",

lastName:"Doe", age:46};

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p>JavaScript uses names to access object properties.</p>
<p id="demo"></p>
<script>
var person = {firstName:"John", lastName:"Doe", age:46};
document.getElementById("demo").innerHTML = person["firstName"];
```

</script>

</body>

</html>

Sample Output

JavaScript Objects

JavaScript uses names to access object properties.

John

• **Array Elements Can Be Objects**

• JavaScript variables can be objects. Arrays are special kinds of objects.

• Because of this, you can have variables of different types in the same Array.

• You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array:

• myArray[0] = Date.now;

myArray[1] = myFunction;

myArray[2] = myCars;

•• **Array Properties and Methods**

• The real strength of JavaScript arrays are the built-in array properties and methods:

• **Examples**

• var x = cars.length; // The length property returns the number of elements

var y = cars.sort(); // The sort() method sorts arrays

- **The length Property**
- The length property of an array returns the length of an array (the number of array elements).
- **Example**
- var fruits =

["Banana", "Orange", "Apple", "Mango"];
fruits.length; // the length of fruits is 4

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The length property returns the length of an array.</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.length;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

The length property returns the length of an array.

4

- **Accessing the First Array Element**
- **Example**
- fruits = ["Banana", "Orange", "Apple", "Mango"];

```
var first = fruits[0];
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var first = fruits[0];
document.getElementById("demo").innerHTML = first;
</script> </body> </html>
```

Sample Output

JavaScript Arrays

JavaScript array elements are accesses using numeric indexes (starting from 0).

Banana

- **Accessing the Last Array Element**
- **Example**
- fruits = ["Banana", "Orange", "Apple", "Mango"];

```
var last = fruits[fruits.length - 1];
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>JavaScript array elements are accesses using numeric indexes (starting from 0).</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var last = fruits[fruits.length-1];
document.getElementById("demo").innerHTML = last;
</script> </body> </html>
```

Sample Output

JavaScript Arrays

JavaScript array elements are accesses using numeric indexes (starting from 0).

Mango

• **Looping Array Elements**

• The safest way to loop through an array, is using a for loop:

• **Example**

```
• var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
text += "<li>" + fruits[i] + "</li>";
}
text += "</ul>";
```

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The best way to loop through an array is using a standard for loop:</p>
<p id="demo"></p>
<script>
var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fLen = fruits.length;
text = "<ul>";
for (i = 0; i < fLen; i++) {
•
text += "<li >" + fruits[i] + "</li>";
}
text += "</ul>";
document.getElementById("demo").innerHTML = text;
</script>
</body>
```

```
</html>
```

Sample Output

JavaScript Arrays

The best way to loop through an array is using a standard for loop:

- Banana
- Orange
- Apple
- Mango

• **You can also use the Array.forEach() function**:

• **Example**

```
• var fruits, text;
fruits = ["Banana", "Orange", "Apple", "Mango"];
text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
function myFunction(value) {
text += "<li>" + value + "</li>";
}<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Array.forEach() calls a function for each array element.</p>
<p>Array.forEach() is not supported in Internet Explorer 8 or earlier.</p>
```

```
<p id="demo"></p>
<script>
var fruits, text;
fruits = ["Banana", "Orange", "Apple", "Mango"];
text = "<ul>";
fruits.forEach(myFunction);
text += "</ul>";
document.getElementById("demo").innerHTML = text;
function myFunction(value) {
text += "<li>" + value + "</li>";
}
</script> </body> </html>
```

Sample Output

JavaScript Arrays

Array.forEach() calls a function for each array element.

Array.forEach() is not supported in Internet Explorer 8 or earlier.

- Banana
- Orange
- Apple
- Mango

• **Adding Array Elements**

• The easiest way to add a new element to an array is using the push() method:

• **Example**

• var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.push("Lemon"); // adds a new element
(Lemon) to fruits

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The push method appends a new element to an array.</p>
<button onclick="myFunction()">Try it</button>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
function myFunction( ) {
fruits.push("Lemon");
document.getElementById("demo").innerHTML = fruits;
}
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

The push method appends a new element to an array.

Try it

Banana,Orange,Apple,Mango

JavaScript Arrays

The push method appends a new element to an array.

Try it

Banana,Orange,Apple,Mango,Lemon

• New element can also be added to an array using the length property:

• **Example**

• var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits[fruits.length] = "Lemon"; // adds a

new element (Lemon) to fruits<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Arrays</h2>

<p>The length property provides an easy way to append new elements to an array without using the push() method.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

```
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits;
function myFunction() {
fruits[fruits.length] = "Lemon";
document.getElementById("demo").innerHTML = fruits;
}
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

The length property provides an easy way to append new elements to an array without using the push() method.

Try it

Banana,Orange,Apple,Mango

JavaScript Arrays

The length property provides an easy way to append new elements to an array without using the push() method.

Try it

Banana,Orange,Apple,Mango,Lemon

• **WARNING !**

• Adding elements with high indexes can create undefined "holes" in an array:

• Example

• var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[6] = "Lemon"; // adds a new element
(Lemon) to fruits

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Adding elements with high indexes can create undefined "holes" in an
array.</p>
<p id="demo"></p>
<script>
var fruits, text, fLen, i;
fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits[6] = "Lemon";
fLen = fruits.length;
text = "";
for (i = 0; i < fLen; i++) {
text += fruits[i] + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Adding elements with high indexes can create undefined "holes" in an array.

Banana
Orange
Apple
Mango
undefined
undefined
Lemon

- **Associative Arrays**
- Many programming languages support arrays with named indexes.
- Arrays with named indexes are called associative arrays (or hashes).
- JavaScript does **not** support arrays with named indexes.
- In JavaScript, **arrays** always use **numbered indexes**.
- **Example**
- var person = [ ];

person[0] = "John";

person[1] = "Doe";

person[2] = 46;

var x = person.length;

// person.length will return 3

var y = person[0];

```
// person[0] will return "John"<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p id="demo"></p>
<script>
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

John 3

- **WARNING !!**

If you use named indexes, JavaScript will redefine the array to a standard object.

After that, some array methods and properties will produce **incorrect results**.

- **Example:**

```
• var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;
// person.length will
return 0
var y = person[0];
// person[0] will return
undefined<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>If you use a named index when accessing an array, JavaScript will redefine the array to a
standard object, and some array methods and properties will produce undefined or incorrect
results.</p>
<p id="demo"></p>
<script>
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
document.getElementById("demo").innerHTML =
person[0] + " " + person.length;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

If you use a named index when accessing an array, JavaScript will redefine the array to a standard object, and some array methods and properties will produce undefined or incorrect results.

undefined 0

- **The Difference Between Arrays and Objects**
- In JavaScript, **arrays** use **numbered indexes**.
- In JavaScript, **objects** use **named indexes**.
- Arrays are a special kind of objects, with numbered indexes.
- **When to Use Arrays. When to use Objects.**
- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers**.
- **Avoid new Array()**
- There is no need to use the JavaScript's built-in array constructor new Array().
- **Use [] instead.**
- These two different statements both create a new empty array named points:
- var points = new Array();

// Bad

var points = [];

// Good

- These two different statements both create a new array containing 6 numbers:
- var points = new Array(40, 100, 1, 5, 25, 10); //

Bad

var points = [40, 100, 1, 5, 25, 10];

// Good

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Avoid using new Array(). Use [] instead.</p>
<p id="demo"></p>
<script>
//var points = new Array(40, 100, 1, 5, 25, 10);
var points = [40, 100, 1, 5, 25, 10];
document.getElementById("demo").innerHTML =
points[0];
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Avoid using new Array(). Use [] instead.

40

• The ***new*** keyword only complicates the code. It can also produce some unexpected results:

• var points = new Array(40, 100); // Creates an array with two elements (40 and 100)

• **What if I remove one of the elements?**

• var points = new Array(40); // Creates an array with 40 undefined elements !!!!!

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>Avoid using new Array().</p>
<p id="demo"></p>
<script>
var points = new Array(40);
document.getElementById("demo").innerHTML =
points[0];
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

Avoid using new Array().

undefined

- **How to Recognize an Array**
- A common question is: How do I know if a variable is an array?
- The problem is that the JavaScript operator typeof returns "object":
- var fruits = ["Banana", "Orange", "Apple", "Mango"];

```
typeof fruits; // returns object<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Arrays</h2>
<p>The typeof operator, when used on an array, returns object:</p>
<p id="demo"></p>
<script>
var fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = typeof fruits;
</script>
</body>
</html>
```

Sample Output

JavaScript Arrays

The typeof operator, when used on an array, returns object:

object

3.11 JavaScript Events

HTML events are **"things"** that happen to HTMLelements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

• Often, when events happen, you may want to do something.

• JavaScript lets you execute code when events are detected.

• HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements

• **With single quotes:**

• <*element event*= **'*some JavaScript*'**>

• **With double quotes:**

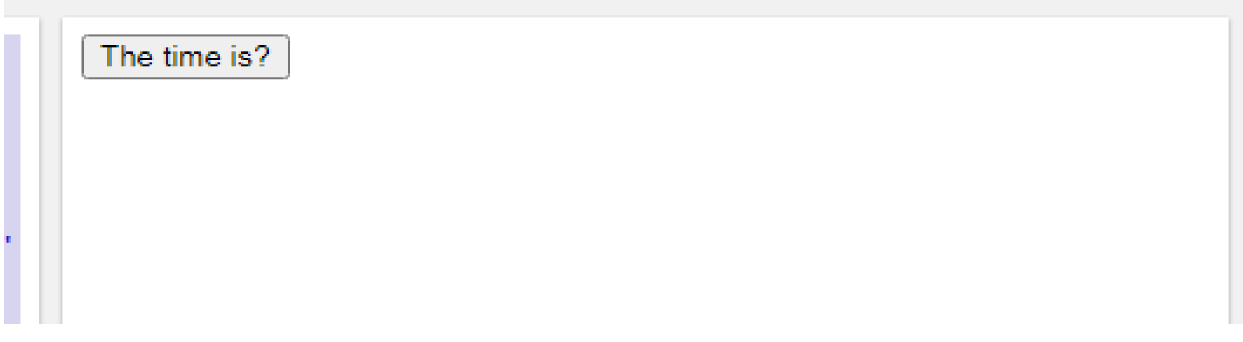
• <*element event*= **"*some JavaScript*"**>

• In the following example, an onclick attribute (with code), is added to a <button> element:

• **Example**

```
<!DOCTYPE html>
<html>
<body>
<button
onclick="document.getElementById('demo').i
nnerHTML=Date()">The time is?</button>
<p id="demo"></p>
</body>
</html>
```

Sample Output



The time is?

Thu Sep 24 2020 16:59:25 GMT+0530 (India Standard Time)

- In the example above, the JavaScript code changes the content of the element with id="demo".
- In the next example, the code changes the content of its own element (using **this**.innerHTML):
- **Example**
- <button onclick="this.innerHTML = Date()">The time is?</button>

```
<!DOCTYPE html>
<html> <body>
<button onclick="this.innerHTML=Date()">The
time is?</button>
</body> </html>
```

Sample Output

The time is?

Thu Sep 24 2020 17:04:44 GMT+0530 (India Standard Time)

- **Example**

```
<!DOCTYPE html>
<html>
<body>
<p>Click the button to display the date.</p>
<button onclick="displayDate()">The time is?</button>
<script>
function displayDate() {
document.getElementById("demo").innerHTML = Date();
}
</script>
<p id="demo"></p>
</body>
</html>
```

Sample Output

Click the button to display the date.

The time is?

Click the button to display the date.

The time is?

Thu Sep 24 2020 17:09:30 GMT+0530 (India Standard Time)

Common HTML Events

Here is a list of some common HTML events:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

• **What can JavaScript Do?**

• Event handlers can be used to handle, and verify, user input, user actions, and browser actions:

• Things that should be done every time a page loads

• Things that should be done when the page is closed

• Action that should be performed when a user clicks a button

• Content that should be verified when a user inputs data

• And more ...

• Many different methods can be used to let JavaScript work with events:

• HTML event attributes can execute JavaScript code directly

• HTML event attributes can call JavaScript functions

• You can assign your own event handler functions to HTML elements

• You can prevent events from being sent or being handled

• And more ...

3.12 JavaScript Objects

Real Life Objects, Properties, and Methods.In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:

All cars have the same **properties**, but the property **values** differ from car to car.

All cars have the same **methods**, but the methods are performed **at different times**.

Object	Properties	Methods
	car.name = Fiat car.model = 500 car.weight = 850kg car.color = white	car.start() car.drive() car.brake() car.stop()

- **JavaScript Objects**
- You have already learned that JavaScript variables are containers for data values.
- This code assigns a **simple value** (Fiat) to a **variable** named car:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Variables</h2>
<p id="demo"></p>
<script>
// Create and display a variable:
var car = "Fiat";
document.getElementById("demo").innerHTML = car;
</script>
</body>
</html>
```

Sample Output

JavaScript Variables

Fiat

• Objects are variables too. But objects can contain many values.

• This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

• var car = {type:"Fiat", model:"500", color:"white"};

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p id="demo"></p>
<script>
// Create an object:
var car = {type:"Fiat", model:"500", color:"white"};
// Display some data from the object:
document.getElementById("demo").innerHTML = "The car type is " +
car.type;
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

The car type is Fiat

The values are written as **name:value** pairs (name and value separated by a colon).

JavaScript objects are containers for **named values** called properties or methods.

- **Object Definition**
- You define (and create) a JavaScript object with an object literal:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p id="demo"></p>
<script>
// Create an object:
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
// Display some data from the object:
document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

John is 50 years old.

• Spaces and line breaks are not important. An object definition can span multiple lines:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p id="demo"></p>
<script>
// Create an object:
var person = {
firstName: "John",
lastName: "Doe",
age: 50,
eyeColor: "blue"
};
// Display some data from the object:
document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

John is 50 years old.

Object Properties

The **name:values** pairs in JavaScript objects are called **properties**:

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

- **Accessing Object Properties**
- You can access object properties in two ways:
- *objectName.propertyName*
- or
- *objectName["propertyName"]*•

Example 1:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p>There are two different ways to access an object property.</p>
<p>You can use person.property or person["property"].</p>
<p id="demo"></p>
<script>
// Create an object:
var person = {
firstName: "John",
lastName : "Doe",
id : 5566
};
// Display some data from the object:
document.getElementById("demo").innerHTML =
person.firstName + " " + person.lastName;
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

There are two different ways to access an object property.

You can use person.property or person["property"].

John Doe

• **Example 2**:

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p>There are two different ways to access an object property.</p>
<p>You can use person.property or person["property"].</p>
<p id="demo"></p>
<script>
// Create an object:
var person = {
firstName: "John",
lastName : "Doe",
id : 5566
};
// Display some data from the object:
```

```
document.getElementById("demo").innerHTML =
person["firstName"] + " " + person["lastName"];
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

There are two different ways to access an object property.

You can use person.property or person["property"].

John Doe

Object Methods

Objects can also have **methods**.

Methods are **actions** that can be performed on objects.

Methods are stored in properties as **function definitions**.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

• A method is a function stored as a property.

• **Example**

• var person = {

firstName: "John",

lastName : "Doe",

id

: 5566,

fullName : function() {

return this.firstName + " " + this.lastName;

}

};

•• The **this** Keyword

• In a function definition, this refers to the "owner" of the function.

• In the example above, this is the **person object** that "owns" the fullName function.

• In other words, this.firstName means the firstName property of **this object**.

• **Accessing Object Methods**

• You access an object method with the following syntax:

• *objectName.methodName()*

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Objects</h2>
<p>An object method is a function definition, stored as a property value.</p>
<p id="demo"></p>
<script>
// Create an object:
var person = {
firstName: "John",
lastName : "Doe",
id : 5566,
```

```
fullName : function() {
return this.firstName + " " + this.lastName;
}
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName();
</script>
</body>
</html>
```

Sample Output

JavaScript Objects

An object method is a function definition, stored as a property value.

John Doe

• If you access a method **without** the () parentheses, it will return the **function definition**:

```
<!DOCTYPE html>
<html> <body>
<h2>JavaScript Objects</h2>
<p>If you access an object method without (), it will return the function
definition:</p>
```

```
<p id="demo"></p>
<script>
// Create an object:
var person = {
firstName: "John",
lastName : "Doe",
id : 5566,
fullName : function() {
return this.firstName + " " + this.lastName;
}
};
// Display data from the object:
document.getElementById("demo").innerHTML = person.fullName;
</script> </body> </html>
```

Sample Output

JavaScript Objects

If you access an object method without (), it will return the function definition:

function() { return this.firstName + " " + this.lastName; }

• **Do Not Declare Strings, Numbers, and Booleans as Objects!**

• When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

• var x = new String();

// Declares x as a String

object

var y = new Number();

// Declares y as a Number object

var z = new Boolean();

// Declares z as a Boolean object

• Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

UNIT-IV

NODE JS: Introduction to Node JS, Installation of Nods JS, Node JS File System Modules, NODE JS npm, Errors, Crypto Module, NODE JS Callbacks, Events, Web Modules. Introduction to Express JS, Express JS Installation, Express JS- GET, POST, REQUEST, RESPONSE, Express JS- Cookies, Middleware Routing.

4.1 What is Node.js?

Node.js is a free, open source tool that lets you run JavaScript outside the web browser.

Node.js is a free, open-source JavaScript runtime that runs on Windows, Mac, Linux, and more. With Node.js, you can build fast and scalable applications like web servers, APIs, tools, and more.

It lets you execute JavaScript code outside of a web browser, enabling server-side development with JavaScript.

Built on Chrome's V8 JavaScript engine, Node.js is designed for building scalable network applications efficiently.

What Can You Build With Node.js?

Node.js uses an **event-driven**, **non-blocking** model. It can handle many connections at once without waiting for one to finish before starting another.This makes it great for real-time apps and high-traffic websites.

Here are some examples of what you can build with Node.js:

- Web servers and websites
- REST APIs
- Real-time apps (like chat) Command-line tools
- Working with files and databases IoT and hardware Control

➢ **How to Run Node.js Code**

Save your code in a file, for example app.js, then run it in your terminal or command prompt with:

```
node app.js
```

Our "Show Node.js" tool makes it easy to learn Node.js

```
let http = require('http');
http.createServer(function (req, res) {
```

```
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World!');
}).listen(8080);
```

SAMPLE OUTPUT:

Hello World!

Why Node.js?

• Node.js excels at handling many simultaneous connections with minimal overhead, making it perfect for:

• **Real-time applications** (chats, gaming, collaboration tools)

• APIs and micro services

• Data streaming applications

• Command-line tools

• Server-side web applications

• Its non-blocking, event-driven architecture makes it highly efficient for I/O-heavy workloads.

• **Asynchronous Programming**

• Node.js uses **asynchronous** (non-blocking) programming.

• This means it can keep working while waiting for tasks like reading files or talking to a database.

• With asynchronous code, Node.js can handle many things at once—making it fast and efficient.

• **Example: Read a File Asynchronously**

```
• // Load the filesystem module
const fs = require('fs');
// Read file asynchronously
fs.readFile('myfile.txt', 'utf8', (err, data) => {
if (err) {
console.error('Error reading file: ' + err);
return;
}
```

```
console.log('File content: ' + data);
});
console.log('Reading file... (this runs first!)');
```

• **SAMPLE OUTPUT:**

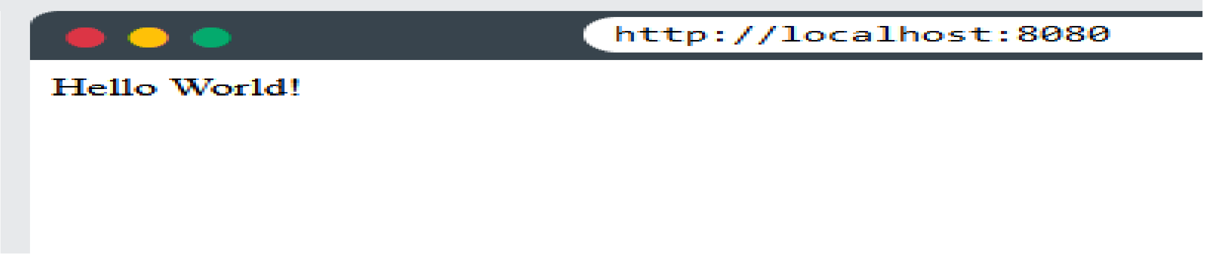
```
Reading file... (this runs first!)
File content: This is the content of myfile.txt
```

What Can Node.js Do?

- **Web Servers**: Create fast, scalable network applications
- **File Operations**: Read, write, and manage files on the server
- **Database Interaction**: Work with databases like MongoDB, MySQL, and more
- **APIs**: Build RESTful services and GraphQLAPIs
- **Real-time**: Handle WebSockets for live applications
- **CLI Tools**: Create command-line applications
- const http = require('http');

```
http.createServer((req, res) => {
res.writeHead(200, {'Content-Type': 'text/plain'});
res.end('Hello World!');
}).listen(8080);
```

• **SAMPLE OUTPUT:**



What is a Node.js File?

• Node.js files contain code that runs on the server. They usually have the .js extension and can be started with the node command.

• Node.js files run tasks when certain events happen (like a web request)

• They must be started on the server to have any effect

• They use JavaScript syntax

• **Node.js Versions & LTS:**

• Node.js releases a new major version every six months.

• For stability, use an **LTS (Long Term Support)** version for production projects.

4.2 Installation of Node JS

Node.js can be installed on various operating systems using different methods. The recommended approach for most users is to use a Node.js installer or a Node Version Manager (NVM).

1. Using a Node.js Installer

Download:

Visit the official Node.js website (nodejs.org) and download the Long Term Support (LTS) version installer for your operating system (Windows, macOS, Linux).

Run the Installer:

Execute the downloaded installer and follow the on-screen instructions. This typically involves accepting the license agreement, choosing an installation location, and selecting components (including

npm, the Node Package Manager, which is usually installed by default).

Verification:

After installation, open your command prompt or terminal and verify the installation by running:

Code:

node –v

npm –v

These commands should display the installed versions of Node.js and npm.

• 2. Using a Node Version Manager (NVM) (Recommended for advanced users and developers):

• NVM allows you to install and manage multiple Node.js versions on your system, which is useful for working on projects requiring different Node.js environments.

• Install NVM:

• Follow the installation instructions specific to your operating system for NVM (e.g., nvm for Linux/macOS, nvm-windows for Windows).

• Install Node.js with NVM:

• Once NVM is installed, you can install a specific Node.js version (e.g., the latest LTS version) using NVM commands:

• Code

• N

• vm install –lts

• Use a Specific Version: You can then switch between installed Node.js versions

• using nvm use <version_number>

4.3 Introduction to Node.js File System

The Node.js File System module (fs) provides a comprehensive set of methods for working with the file system on your computer.

It allows you to perform file I/O operations in both synchronous and asynchronous ways.

Importing the File System Module

You can import the File System module using CommonJS require( ) or ES modules import syntax:

CommonJS (Default in Node.js)

```
const fs = require('fs');
```

ES Modules (Node.js 14+ with "type": "module" in package.json)

```
import fs from 'fs';

// Or for specific methods:

// import { readFile, writeFile } from 'fs/promises';
```

Promise-based API

Node.js provides promise-based versions of the File System API in the fs/promises namespace, which is recommended for modern applications:

```
// Using promises (Node.js 10.0.0+)

const fs = require('fs').promises;

// Or with destructuring

const { readFile, writeFile } = require('fs').promises;

// Or with ES modules

// import { readFile, writeFile } from 'fs/promises';
```

- **Common Use Cases**

File Operations

- Read and write files
- Create and delete files
- Append to files
- Rename and move files
- Change file permissions

Directory Operations

• Create and remove directories

• List directory contents

• Watch for file changes

• Get file/directory stats

• Check file existence

Advanced Features

• File streams

• File descriptors

• Symbolic links

• File watching

• Working with file permissions

• Reading Files

• Node.js provides several methods to read files, including both callback-based and promise-based approaches.

• The most common method is

• fs.readFile( );

• Note: Always handle errors when working with file operations to prevent your application from crashing.

• Reading Files with Callbacks

• Here's how to read a file using the traditional callback pattern:

• myfile.txt

• This is the content of myfile.txt

• Create a Node.js file that reads the text file, and return the content:

• Example: Reading a file with callbacks

• const fs = require('fs');

```
// Read file asynchronously with callback
fs.readFile('myfile.txt', 'utf8', (err, data) => {
if (err) {
```

```
console.error('Error reading file:', err);
return;
}
console.log('File content:', data);
});
// For binary data (like images), omit the encoding
fs.readFile('image.png', (err, data) => {
if (err) throw err;
// data is a Buffer containing the file content
console.log('Image size:', data.length, 'bytes');
});
```

- **Sample Output:**

```
Image size: 39709 bytes
File content: This is the content of myfile.txt
```

- **Reading Files with Promises (Modern Approach)**
- Using fs.promises( ) or util.promisify( ) for cleaner async/await syntax:
- **Example: Reading a file with async/await**
- // Using fs.promises (Node.js 10.0.0+)

```
const fs = require('fs').promises;
async function readFileExample() {
try {
const data = await fs.readFile('myfile.txt', 'utf8');
console.log('File content:', data);
} catch (err) {
```

```
console.error('Error reading file:', err);
} }
readFileExample();
// Or with util.promisify (Node.js 8.0.0+)
const { promisify } = require('util');
const readFileAsync = promisify(require('fs').readFile);
async function readWithPromisify() {
try {
const data = await readFileAsync('myfile.txt', 'utf8');
console.log(data);
} catch (err) {
console.error(err);
} }
readWithPromisify();
```

• **Sample Output:**

```
File content: This is the content of myfile.txt
This is the content of myfile.txt
```

• **Reading Files Synchronously**

• For simple scripts, you can use synchronous methods, but avoid them in production servers as they block the event loop:

• **Example: Reading a file synchronously**

```
• const fs = require('fs');
try {
// Read file synchronously
const data = fs.readFileSync('myfile.txt', 'utf8');
```

```
console.log('File content:', data);
} catch (err) {
console.error('Error reading file:', err);
}
```

• **Best Practice:**

Always specify the character encoding (like 'utf8') when reading text files to get a string instead of a Buffer.

• Creating and Writing Files

• Node.js provides several methods for creating and writing to files.

• Here are the most common approaches:

• **1. Using fs.writeFile( )**

• Creates a new file or overwrites an existing file with the specified content:

• **Example: Writing to a file**

```
• const fs = require('fs').promises;
async function writeFileExample() {
try {
// Write text to a file
await fs.writeFile('myfile.txt', 'Hello, World!', 'utf8');
// Write JSON data
const data = { name: 'John', age: 30, city: 'New York' };
await fs.writeFile('data.json', JSON.stringify(data, null, 2), 'utf8');
console.log('Files created successfully');
} catch (err) {
console.error('Error writing files:', err);
} }
writeFileExample();
```

• Sample Output:

```
Files created successfully
```

• 2. Using fs.appendFile( )

• Appends content to a file, creating the file if it doesn't exist:

• Example: Appending to a file

• const fs = require('fs').promises;

```
async function appendToFile() {
try {
// Append a timestamped log entry
const logEntry = `${new Date().toISOString()}: Application started\n`;
await fs.appendFile('app.log', logEntry, 'utf8');
console.log('Log entry added');
} catch (err) {
console.error('Error appending to file:', err);
}
}
appendToFile();
```

• Sample Output:

```
Log entry added
```

• For more control over file operations, you can use file handles:

• Example: Using file handles

• const fs = require('fs').promises;

```
async function writeWithFileHandle() {
let fileHandle;
try {
// Open a file for writing (creates if doesn't exist)
fileHandle = await fs.open('output.txt', 'w');
// Write content to the file
await fileHandle.write('First line\n');
await fileHandle.write('Second line\n');
await fileHandle.write('Third line\n');
console.log('Content written successfully');
} catch (err) {
console.error('Error writing to file:', err);
} finally {
// Always close the file handle
if (fileHandle) {
await fileHandle.close();
} } }
writeWithFileHandle();
```

• Sample Output:

```
Content written successfully
```

• 4. Using Streams for Large Files

• For writing large amounts of data, use streams to avoid high memory usage:

• Example: Writing large files with streams

```
• const fs = require('fs');
const { pipeline } = require('stream/promises');
const { Readable } = require('stream');
async function writeLargeFile() {
// Create a readable stream (could be from HTTP request, etc.)
const data = Array(1000).fill().map((_, i) => `Line ${i + 1}: ${'x'.repeat(100)}\n`);
const readable = Readable.from(data);
// Create a writable stream to a file
const writable = fs.createWriteStream('large-file.txt');
try {
// Pipe the data from readable to writable
await pipeline(readable, writable);
console.log('Large file written successfully');
} catch (err) {
console.error('Error writing file:', err);
}}
writeLargeFile();
```

• Sample Output

```
Large file written successfully
```

File Flags: When opening files, you can specify different modes:

• ‘w’ - Open for writing (file is created or truncated)

• ‘wx’ - Like 'w' but fails if the path exists

• ‘w+’ - Open for reading and writing (file is created or truncated)

• ‘a’ - Open for appending (file is created if it doesn't exist)

• ‘ax’ - Like 'a' but fails if the path exists

• ‘r+’ - Open for reading and writing (file must exist)

• Deleting Files and Directories

• Node.js provides several methods to delete files and directories.

• Here's how to handle different deletion scenarios:

• 1. Deleting a Single File

• Use fs.unlink( ) to delete a file:

• Example: Deleting a file

```
• const fs = require('fs').promises;
async function deleteFile() {
const filePath = 'file-to-delete.txt';
try {
// Check if file exists before deleting
await fs.access(filePath);
```

```
// Delete the file
await fs.unlink(filePath);
console.log('File deleted successfully');
} catch (err) {
if (err.code === 'ENOENT') {
console.log('File does not exist');
} else {
console.error('Error deleting file:', err);
} }}
deleteFile();
```

• Sample Output:

```
File deleted successfully
```

2) Deleting Multiple Files

• To delete multiple files, you can use Promise.all() with fs.unlink():

• Example: Deleting multiple files

• const fs = require('fs').promises;

```
const path = require('path');

async function deleteFiles() {

const filesToDelete = [

'temp1.txt',

'temp2.txt',
```

```
'temp3.txt'
];
try {
// Delete all files in parallel
await Promise.all(
filesToDelete.map(file =>
fs.unlink(file).catch(err => {
if (err.code !== 'ENOENT') {
console.error(`Error deleting ${file}:`, err);
} })
) );
console.log('Files deleted successfully');
} catch (err) {
console.error('Error during file deletion:', err);
}}
deleteFiles();
```

• **Sample Output:**

```
Directory deleted successfully
```

3. Deleting Directories

• To delete directories, you have several options depending on your needs:

• Example: Deleting directories

```
• const fs = require('fs').promises;

const path = require('path');

async function deleteDirectory(dirPath) {

try {

// Check if the directory exists

const stats = await fs.stat(dirPath);

if (!stats.isDirectory()) {

console.log('Path is not a directory');

return;

}

// For Node.js 14.14.0+ (recommended)

await fs.rm(dirPath, { recursive: true, force: true });

// For older Node.js versions (deprecated but still works)

// await fs.rmdir(dirPath, { recursive: true });

console.log('Directory deleted successfully');

} catch (err) {

if (err.code === 'ENOENT') {

console.log('Directory does not exist');

} else {

console.error('Error deleting directory:', err);
```

```
} }}

// Usage

deleteDirectory('directory-to-delete');
```

Sample Output:

```
Directory deleted successfully
```

4. Emptying a Directory Without Deleting It

• To remove all files and subdirectories within a directory but keep the directory itself:

• Example: Emptying a directory

```
• const fs = require('fs').promises;

const path = require('path');

async function emptyDirectory(dirPath) {

try {

// Read the directory

const files = await fs.readdir(dirPath, { withFileTypes: true });

// Delete all files and directories in parallel

await Promise.all(

files.map(file => {

const fullPath = path.join(dirPath, file.name);

return file.isDirectory()

? fs.rm(fullPath, { recursive: true, force: true })
```

```
: fs.unlink(fullPath);
})
);
console.log('Directory emptied successfully');
} catch (err) {
console.error('Error emptying directory:', err);
} }
// Usage
emptyDirectory('directory-to-empty');
```

• Sample Output:

```
Directory emptied successfully
```

• The fs.rename( ) method can be used for both renaming and moving files.

• It's a versatile method for file system operations that involve changing file paths.

• 1. Basic File Renaming

• To rename a file in the same directory:

• Example: Renaming a file

```
• const fs = require('fs').promises;
async function renameFile() {
const oldPath = 'old-name.txt';
const newPath = 'new-name.txt';
```

```
try {
// Check if source file exists
await fs.access(oldPath);
// Check if destination file already exists
try {
await fs.access(newPath);
console.log('Destination file already exists');
return;
• } catch (err) {
// Destination doesn't exist, safe to proceed
}
• // Perform the rename
await fs.rename(oldPath, newPath);
console.log('File renamed successfully');
} catch (err) {
if (err.code === 'ENOENT') {
console.log('Source file does not exist');
} else {
console.error('Error renaming file:', err);
}
}
}
// Usage
```

renameFile();

• Sample Output:

```
File renamed successfully
```

2. Moving Files Between Directories

• You can use fs.rename( ) to move files between directories:

• Example: Moving a file to a different directory

• const fs = require('fs').promises;

const path = require('path');

async function moveFile() {

const sourceFile = 'source/file.txt';

const targetDir = 'destination';

const targetFile = path.join(targetDir, 'file.txt');

try {

// Ensure source file exists

await fs.access(sourceFile);

// Create target directory if it doesn't exist

await fs.mkdir(targetDir, { recursive: true });

// Move the file

```
await fs.rename(sourceFile, targetFile);
• console.log('File moved successfully');
} catch (err) {
if (err.code === 'ENOENT') {
console.log('Source file does not exist');
} else if (err.code === 'EXDEV') {
• console.log('Cross-device move detected, using copy+delete fallback');
await moveAcrossDevices(sourceFile, targetFile);
} else {
console.error('Error moving file:', err);
}
}
}
• // Helper function for cross-device moves
async function moveAcrossDevices(source, target) {
try {
// Copy the file
await fs.copyFile(source, target);
// Delete the original
await fs.unlink(source);
console.log('File moved across devices successfully');
} catch (err) {
// Clean up if something went wrong
```

try { await fs.unlink(target); } catch (e) {}

throw err;

}

}

// Usage

moveFile();

• Sample Output:

```
File moved successfully
```

3. Batch Renaming Files

• To rename multiple files matching a pattern:

• Example: Batch renaming files

• const fs = require('fs').promises;

const path = require('path');

async function batchRename() {

const directory = 'images';

const pattern = /^image(\d+)\.jpg$/;

try {

```
// Read directory contents

const files = await fs.readdir(directory);

// Process each file

for (const file of files) {

const match = file.match(pattern);

if (match) {

const [_, number] = match;

const newName = `photo-${number.padStart(3, '0')}.jpg`;

const oldPath = path.join(directory, file);

const newPath = path.join(directory, newName);

• // Skip if the new name is the same as the old name

if (oldPath !== newPath) {

await fs.rename(oldPath, newPath);

console.log(`Renamed: ${file} - ${newName}`);

}

}

}

console.log('Batch rename completed');

} catch (err) {

console.error('Error during batch rename:', err);

}

}

batchRename();
```

• **Sample Output:**

```
Renamed: image3211.jpg - photo-3211.jpg
Renamed: image3212.jpg - photo-3212.jpg
Renamed: image3213.jpg - photo-3213.jpg
Renamed: image3214.jpg - photo-3214.jpg
Renamed: image3215.jpg - photo-3215.jpg
Renamed: image3216.jpg - photo-3216.jpg
Renamed: image3217.jpg - photo-3217.jpg
Renamed: image3218.jpg - photo-3218.jpg
Batch rename completed
```

4. Atomic Rename Operations

• For critical operations, use a temporary file to ensure atomicity:

• **Example: Atomic file update**

• const fs = require('fs').promises;

const path = require('path');

const os = require('os');

async function updateFileAtomic(filePath, newContent) {

const tempPath = path.join(

os.tmpdir(),

• `tem

```
• );
try {
// 1. Write to temp file
await fs.writeFile(tempPath, newContent, 'utf8');
// 2. Verify the temp file was written correctly
const stats = await fs.stat(tempPath);
if (stats.size === 0) {
throw new Error('Temporary file is empty');
}p-${Date.now()}-${Math.random().toString(36).substr(2, 9)}`
• // 3. Rename (atomic on most systems)
await fs.rename(tempPath, filePath);
console.log('File updated atomically');
} catch (err) {
// Clean up temp file if it exists
try { await fs.unlink(tempPath); } catch (e) {}
console.error('Atomic update failed:', err);
throw err;
}
}
// Usage
updateFileAtomic('important-config.json', JSON.stringify({ key: 'value' }, null,
2));
```

4.4 Node.js NPM

• What is NPM?

NPM is a package manager for Node.js packages, or modules if you like.www.npmjs.com hosts thousands of free packages to download and use.The NPM program is installed on your computer when you install Node.js

• What is a Package?

A package in Node.js contains all the files you need for a module.Modules are JavaScript libraries you can include in your project.

• Download a Package

Downloading a package is very easy.

Open the command line interface and tell NPM to download the package you want.

I want to download a package called "upper-case":

Download "upper-case":

C:\Users\Your Name>npm install upper-case

• Now you have downloaded and installed your first package!

• NPM creates a folder named "node_modules", where the package will be placed.

• All packages you install in the future will be placed in this folder.

• My project now has a folder structure like this:

• C:\Users\My Name\node_modules\upper-case

• Using a Package

• Once the package is installed, it is ready to use.

• Include the "upper-case" package the same way you include any other module:

• let uc = require('upper-case');

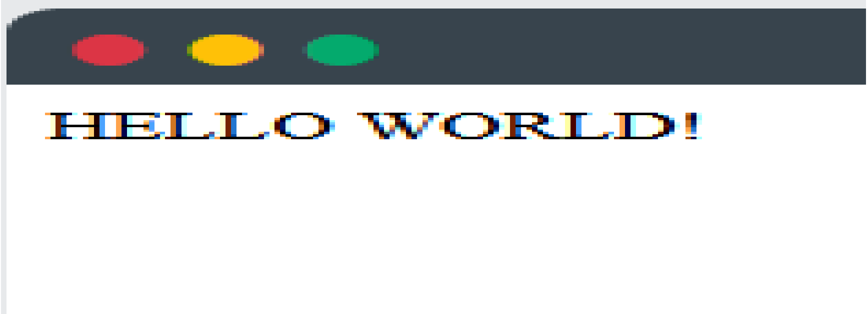
• Create a Node.js file that will convert the output "Hello World!" into upper-case letters:

• Example : Get your own Node.js Server

• let http = require('http');

```
let uc = require('upper-case');
http.createServer(function (req, res) {
res.writeHead(200, {'Content-Type': 'text/html'});
res.write(uc.upperCase("Hello World!"));
res.end();
}).listen(8080);
```

• Sample Output:



• Save the code above in a file called "demo_uppercase.js", and initiate the file:

• Initiate demo_uppercase:

• C:\Users\Your Name>node demo_uppercase.js

• If you have followed the same steps on your computer, you will see the same result as the example: http://localhost:8080

• Global Packages

• Packages can be installed globally, making them available as command-line tools anywhere on your system.

• Global packages are typically used for CLI tools and utilities.

• Install a package globally:
• npm install -g package-name
• Example: Install the http-server package globally
• npm install -g http-server
• After installation, you can run the package from any directory:
• http-server
• Updating Packages
• To keep your packages up to date, you can update them using the following commands:
Update a specific package:
• npm update package-name
Update all packages in your project:
• npm update
Check for outdated packages:
• npm outdated
• Uninstalling a Package
• To remove a package that you no longer need, you can use the uninstall command:
• Remove a package:
• npm uninstall package-name
• Remove a global package:
• npm uninstall -g package-name
• Remove a package and its dependencies:
• npm uninstall --save package-name

4.5 Node.js Error Handling

Why Handle Errors?

Errors are inevitable in any program, but how you handle them makes all the difference. In Node.js,proper error handling is crucial because:

It prevents applications from crashing unexpectedly It provides meaningful feedback to users

It makes debugging easier with proper error context . It helps maintain application stability in production. It ensures resources are properly cleaned up.

Common Error Types in Node.js

Understanding different error types helps in handling them appropriately:

1. Standard JavaScript Errors

```
// SyntaxError

JSON.parse('{invalid json}');

// TypeError

null.someProperty;

// ReferenceError

unknownVariable;
```

• 2. System Errors

```
• // ENOENT: No such file or directory

const fs = require('fs');

fs.readFile('nonexistent.txt', (err) => {

console.error(err.code); // 'ENOENT'

});

// ECONNREFUSED: Connection refused

const http = require('http');

const req = http.get('http://nonexistent-site.com', (res) => {});

req.on('error', (err) => {

console.error(err.code); // 'ECONNREFUSED' or 'ENOTFOUND'

});
```

• **Basic Error Handling**

• Node.js follows several patterns for error handling:

• **Error-First Callbacks**

• The most common pattern in Node.js core modules where the first argument to a callback is an error object (if any occurred).

• **Example: Error-First Callback**

• const fs = require('fs');

```
function readConfigFile(filename, callback) {
fs.readFile(filename, 'utf8', (err, data) => {
if (err) {
// Handle specific error types
if (err.code === 'ENOENT') {
return callback(new Error(`Config file ${filename} not found`));
} else if (err.code === 'EACCES') {
return callback(new Error(`No permission to read ${filename}`));
}
// For all other errors
return callback(err);
}
// Process data if no error
try {
const config = JSON.parse(data);
callback(null, config);
```

} catch (parseError) {

callback(new Error(`Invalid JSON in ${filename}`));

}

});

}

• // Usage

readConfigFile('config.json', (err, config) => {

if (err) {

console.error('Failed to read config:', err.message);

// Handle the error (e.g., use default config)

return;

}

console.log('Config loaded successfully:', config);

});

• **Sample Output:**

```
Failed to read config: Config file config.json not found
```

Modern Error Handling

- Using try...catch with Async/Await
- With async/await, you can use try/catch blocks for both synchronous and asynchronous code:
- Example: try/catch with Async/Await
- const fs = require('fs').promises;

```
async function loadUserData(userId) {

try {

const data = await fs.readFile(`users/${userId}.json`, 'utf8');

const user = JSON.parse(data);

if (!user.email) {

throw new Error('Invalid user data: missing email');

}

return user;

} catch (error) {

// Handle different error types

if (error.code === 'ENOENT') {

throw new Error(`User ${userId} not found`);

} else if (error instanceof SyntaxError) {

• throw new Error('Invalid user data format');

}

// Re-throw other errors

throw error;

} finally {
```

```
// Cleanup code that runs whether successful or not
console.log(`Finished processing user ${userId}`);
}
}
// Usage
(async () => {
try {
const user = await loadUserData(123);
console.log('User loaded:', user);
} catch (error) {
console.error('Failed to load user:', error.message);
// Handle error (e.g., show to user, retry, etc.)
}
})();
```

• **Sample Output:**

```
Finished processing user 123
Failed to load user: User 123 not found
```

• **Uncaught Exceptions**

• For unexpected errors, you can listen for uncaughtException to perform cleanup before exiting:

• Example: Global Error Handlers

• // Handle uncaught exceptions (synchronous errors)

```
process.on('uncaughtException', (error) => {
console.error('UNCAUGHT EXCEPTION! Shutting down...');
console.error(error.name, error.message);
// Perform cleanup (close database connections, etc.)
server.close(() => {
console.log('Process terminated due to uncaught exception');
process.exit(1); // Exit with failure
});
});
```

• // Handle unhandled promise rejections

```
process.on('unhandledRejection', (reason, promise) => {
console.error('UNHANDLED REJECTION! Shutting down...');
console.error('Unhandled Rejection at:', promise, 'Reason:', reason);
// Close server and exit
server.close(() => {
process.exit(1);
});
});
// Example of an unhandled promise rejection
```

```
Promise.reject(new Error('Something went wrong'));

// Example of an uncaught exception

setTimeout(() => {

throw new Error('Uncaught exception after timeout');

}, 1000);
```

- **Error Handling Best Practices**
- **Dos and Don'ts**
- **Do**
- Handle errors at the appropriate level
- Log errors with sufficient context
- Use custom error types for different scenarios
- Clean up resources in finally blocks
- Validate input to catch errors early
- **Don't**
- Ignore errors (empty catch blocks)
- Expose sensitive error details to clients
- Use try/catch for flow control
- Swallow errors without logging them
- Continue execution after unrecoverable errors.

4.6 Introduction to Node.js Crypto Module

What is the Crypto Module?

The Crypto module is a built-in Node.js module that provides cryptographic functionality including:

- Hash functions (SHA-256, SHA-512, etc.)
- HMAC (Hash-based Message Authentication Code)
- Symmetric encryption (AES, DES, etc.)
- Asymmetric encryption (RSA, ECDSA, etc.)
- Digital signatures and verification
- Secure random number generation

The Crypto module is essential for applications that need to handle sensitive information securely.

The Crypto module wraps the OpenSSL library, providing access to well-established and tested cryptographic algorithms.

This module is often used to handle sensitive data, such as:

- User authentication and password storage
- Secure data transmission
- File encryption and decryption
- Secure communication channels
- Getting Started with Crypto
- Here's a quick example of using the Crypto module to hash a string:
- const crypto = require('crypto');

```
// Create a SHA-256 hash of a string

const hash = crypto.createHash('sha256')

.update('Hello, Node.js!')

.digest('hex');

console.log('SHA-256 Hash:', hash);
```

```
SHA-256 Hash: 4695813a8d1ade632d4bcf74d2e58e296883db03306149e8c78f2de0886ef665
```

Installing the Crypto Module

- The Crypto module is included in Node.js by default.
- You can use it by requiring it in your script:
- const crypto = require('crypto');

Hash Functions

- Hashing is a one-way transformation of data into a fixed-length string of characters.

Hash functions have several important properties:

- Deterministic: Same input always produces the same output
- Fixed Length: Output is always the same size regardless of input size
- One-Way: Extremely difficult to reverse the process
- Avalanche Effect: Small changes in input produce significant changes in output

Common use cases include:

• Password storage

• Data integrity verification

• Digital signatures

• Content addressing (e.g., Git, IPFS)

• Creating a Hash

• const crypto = require('crypto');

```
// Create a hash object
const hash = crypto.createHash('sha256');
// Update the hash with data
hash.update('Hello, World!');
// Get the digest in hexadecimal format
const digest = hash.digest('hex');
console.log(digest);
```

```
dffd6021bb2bd5b0af676655809ec3a53191dd81c7f70a4b28688a362182986f
```

Common Hash Algorithms

```
• const crypto = require('crypto');

const data = 'Hello, World!';

// MD5 (not recommended for security-critical applications)

const md5 = crypto.createHash('md5').update(data).digest('hex');

console.log('MD5:', md5);

// SHA-1 (not recommended for security-critical applications)

const sha1 = crypto.createHash('sha1').update(data).digest('hex');

console.log('SHA-1:', sha1);

// SHA-256

const sha256 = crypto.createHash('sha256').update(data).digest('hex');

console.log('SHA-256:', sha256);

// SHA-512

const sha512 = crypto.createHash('sha512').update(data).digest('hex');

console.log('SHA-512:', sha512);
```

```
MD5: 65a8e27d8879283831b664bd8b7f0ad4
SHA-1: 0a0a9f2a6772942557ab5355d76af442f8f65e01
SHA-256: dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f
SHA-512: 374d794a95cdcfd8b35993185fef9ba368f160d8daf432d08ba9f1ed1e5abe6cc69291e0fa2fe0006a52570ef18c19de
```

• **Password Security**

• When handling passwords, it's crucial to use specialized password hashing functions that are designed to be computationally expensive to prevent brute-force attacks.

• Here's why simple hashes are insufficient:

• Never store passwords in plain text or with simple hashes like MD5 or SHA-1.

• These can be easily cracked using rainbow tables or brute-force attacks.

• Key Concepts for Password Security

• **Salting:** Add a unique random value to each password before hashing

• **Key Stretching:** Make the hashing process intentionally slow to prevent brute-force attacks

• **Work Factor:** Control how computationally intensive the hashing process is

• Here's how to properly hash passwords in Node.js:

• **What is a salt?**

A salt is a random string that is unique to each user.

• It's combined with the password before hashing to ensure that even if two users have the same password, their hashes will be different.

• This prevents attackers from using precomputed tables (like rainbow tables) to crack multiple passwords at once.

```
• const crypto = require('crypto');
// Function to hash a password
function hashPassword(password) {
// Generate a random salt (16 bytes)
const salt = crypto.randomBytes(16).toString('hex');
```

```
// Use scrypt for password hashing (recommended)

const hash = crypto.scryptSync(password, salt, 64).toString('hex');

// Return both salt and hash for storage

return { salt, hash };

}

// Function to verify a password

function verifyPassword(password, salt, hash) {

const hashedPassword = crypto.scryptSync(password, salt, 64).toString('hex');

return hashedPassword === hash;

}

// Example usage

const password = 'mySecurePassword';

// Hash the password for storage

const { salt, hash } = hashPassword(password);

console.log('Salt:', salt);

console.log('Hash:', hash);

// Verify a login attempt

const isValid = verifyPassword(password, salt, hash);

console.log('Password valid:', isValid); // true

const isInvalid = verifyPassword('wrongPassword', salt, hash);

console.log('Wrong password valid:', isInvalid); // false
```

```
Salt: 07691f6ed938a064f88a6c485d690fff
Hash: 929c945eee7751f21b6e0ac3f2d7f571e477df6f15aba617b3e145ea423a99980a35f3a7b6a3ce5fb215b6679e6fc3672104
Password valid: true
Wrong password valid: false
```

• HMAC (Hash-based Message Authentication Code)

• HMAC is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key.

• It provides both data integrity and authentication.

• When to Use HMAC

• API request verification

• Secure cookies and sessions

• Data integrity checks

• Webhook verification

• HMAC Security Properties

• Message Integrity: Any change to the message will produce a different HMAC

• Authenticity: Only parties with the secret key can generate valid HMACs

• No Encryption: HMAC doesn't encrypt the message, only verifies its integrity

• const crypto = require('crypto');

// Secret key

const secretKey = 'mySecretKey';

// Create an HMAC

const hmac = crypto.createHmac('sha256', secretKey);

// Update with data

hmac.update('Hello, World!');

// Get the digest

const hmacDigest = hmac.digest('hex');

console.l

• og('HMAC:', hmacDigest

```
HMAC: 90a39237924dcbbc7a2ea3bf5bcc7aa6debdf67d95d783306b2550ca2da64b2d
```

HMAC for Message Verification

• const crypto = require('crypto');

// Function to create an HMAC for a message

function createSignature(message, key) {

const hmac = crypto.createHmac('sha256', key);

hmac.update(message);

return hmac.digest('hex');

}

// Function to verify a message's signature

function verifySignature(message, signature, key) {

const expectedSignature = createSignature(message, key);

return crypto.timingSafeEqual(

Buffer.from(signature, 'hex'),

Buffer.from(expectedSignature, 'hex')

);

}

// Example usage

const secretKey = 'verySecretKey';

const message = 'Important message to verify';

• // Sender creates a signature

const signature = createSignature(message, secretKey);

console.log('Message:', message);

console.log('Signature:', signature);

```
// Receiver verifies the signature

try {

const isValid = verifySignature(message, signature, secretKey);

console.log('Signature valid:', isValid); // true

// Try with a tampered message

const isInvalid = verifySignature('Tampered message', signature, secretKey);

console.log('Tampered message valid:', isInvalid); // false

} catch (error) {

console.error('Verification error:', error.message);
```

```
Message: Important message to verify
Signature: a6940691f50ccc6c6424e1d79af453b7e139d3bc7c0046494a733d08daeecd84
Signature valid: true
Tampered message valid: false
```

• Symmetric encryption uses the same key for both encryption and decryption.

• It's generally faster than asymmetric encryption and is ideal for:

• Bulk data encryption

• Database encryption

• Filesystem encryption

• Secure messaging (combined with key exchange)

Common Symmetric Algorithms

Algorithm	Key Size	Block Size	Notes
AES-256	256 bits	128 bits	Current standard, widely used
ChaCha20	256 bits	512 bits	Faster in software, used in TLS 1.3
3DES	168 bits	64 bits	Legacy, not recommended for new systems
Blowfish	32-448 bits	64 bits	Legacy, use Twofish or AES instead

AES (Advanced Encryption Standard)

```
• const crypto = require('crypto');
// Function to encrypt data
function encrypt(text, key) {
// Generate a random initialization vector
const iv = crypto.randomBytes(16);
// Create cipher with AES-256-CBC
const cipher = crypto.createCipheriv('aes-256-cbc', key, iv);
// Encrypt the data
let encrypted = cipher.update(text, 'utf8', 'hex');
encrypted += cipher.final('hex');
// Return both the encrypted data and the IV
return {
iv: iv.toString('hex'),
```

```
encryptedData: encrypted
};
}
// Function to decrypt data
function decrypt(encryptedData, iv, key) {
// Create decipher
const decipher = crypto.createDecipheriv(
'aes-256-cbc',
key,
Buffer.from(iv, 'hex')
);
• // Decrypt the data
let decrypted = decipher.update(encryptedData, 'hex', 'utf8');
decrypted += decipher.final('utf8');
return decrypted;
}
// Example usage
// Note: In a real application, use a properly generated and securely stored key
const key = crypto.scryptSync('secretPassword', 'salt', 32); // 32 bytes = 256 bits
const message = 'This is a secret message';
// Encrypt
const { iv, encryptedData } = encrypt(message, key);
console.log('Original:', message);
```

```
console.log('Encrypted:', encryptedData);

console.log('IV:', iv);

// Decrypt

const decrypted = decrypt(encryptedData, iv, key);

console.log('Decrypted:', decrypted);
```

```
Original: This is a secret message
Encrypted: 32bab4c50435e46f951cf1c0b1ff6d9b0d96f9032198746bbc463c761f4371d7
IV: 003e4fbd5235013d7615be8ac5987e66
Decrypted: This is a secret message
```

Other Symmetric Algorithms

- The Crypto module supports various symmetric encryption algorithms.
- You can see the available ciphers with:
- const crypto = require('crypto');

```
// List available cipher algorithms

console.log(crypto.getCiphers());
```

```
[
  'aes-128-cbc',
  'aes-128-cbc-hmac-sha1',
  'aes-128-cbc-hmac-sha256',
  'aes-128-ccm',
  'aes-128-cfb',
  'aes-128-cfb1',
  'aes-128-cfb8',
  'aes-128-ctr',
  'aes-128-ecb',
  'aes-128-gcm',
  'aes-128-ocb',
  'aes-128-ofb',
  'aes-128-xts',
  'aes-192-cbc',
  'aes-192-ccm',
  'aes-192-cfb',
  'aes-192-cfb1',
  'aes-192-cfb8',
  'aes-192-ctr',
  'aes-192-ecb',
  'aes-192-gcm',
  'aes-192-ocb',
  'aes-192-ofb',
  'aes-256-cbc',
  'aes-256-cbc-hmac-sha1',
  'aes-256-cbc-hmac-sha256',
  'aes-256-ccm',
  'aes-256-cfb',
  'aes-256-cfb1',
  'aes-256-cfb8',
  'aes-256-ctr',
```

Asymmetric Encryption

• Asymmetric encryption (public-key cryptography) uses a pair of mathematically related keys:

• **Public Key**: Can be shared publicly, used for encryption

• **Private Key**: Must be kept secret, used for decryption

• **Common Use Cases**

• Secure key exchange (e.g., TLS/SSL handshake)

• Digital signatures

• Email encryption (PGP/GPG)

• Blockchain and cryptocurrencies

Common Asymmetric Algorithms

Algorithm	Key Size	Security Level	Notes
RSA	2048+ bits	High	Widely used, good compatibility
ECDSA	256-521 bits	High	Used in TLS 1.3, Bitcoin
Ed25519	256 bits	Very High	Modern, efficient, used in SSH

Performance Note:

- Asymmetric encryption is much slower than symmetric encryption.
- For encrypting large amounts of data, use a hybrid approach:
- Generate a random symmetric key
- Encrypt your data with the symmetric key
- Encrypt the symmetric key with the recipient's public key
- Send both the encrypted data and encrypted key.

- **RSA (Rivest-Shamir-Adleman)**

```
• const crypto = require('crypto');
// Generate RSA key pair
function generateKeyPair() {
return crypto.generateKeyPairSync('rsa', {
modulusLength: 2048, // Key size in bits
publicKeyEncoding: {
type: 'spki',
```

```
format: 'pem'
},
privateKeyEncoding: {
type: 'pkcs8',
format: 'pem'
}
});
}
// Encrypt with public key
function encryptWithPublicKey(text, publicKey) {
const buffer = Buffer.from(text, 'utf8');
const encrypted = crypto.publicEncrypt(
{
key: publicKey,
padding: crypto.constants.RSA_PKCS1_OAEP_PADDING
},
buffer
); return encrypted.toString('base64');
}
• // Decrypt with private key
function decryptWithPrivateKey(encryptedText, privateKey) {
const buffer = Buffer.from(encryptedText, 'base64');
const decrypted = crypto.privateDecrypt(
```

```
{
key: privateKey,
padding: crypto.constants.RSA_PKCS1_OAEP_PADDING
},
buffer
);
return decrypted.toString('utf8');
}
// Generate keys
const { publicKey, privateKey } = generateKeyPair();
console.log('Public Key:', publicKey.substring(0, 50) + '...');
console.log('Private Key:', privateKey.substring(0, 50) + '...');
// Example usage
const message = 'This message is encrypted with RSA';
const encrypted = encryptWithPublicKey(message, publicKey);
console.log('Encrypted:', encrypted.substring(0, 50) + '...');
const decrypted = decryptWithPrivateKey(encrypted, privateKey);
console.log('Decrypted:', decrypted);
```

```
Public Key: -----BEGIN PUBLIC KEY-----
MIOKIjANBgkqhkiG9w0BAQE...
Private Key: -----BEGIN PRIVATE KEY-----
MIIEvQIKHUANBgkqhkiG9w...
Encrypted: RsDXpvjB71/dP2QgA5PuHiKC42YgmY...
Decrypted: This message is encrypted with RSA
```

- **Digital Signatures**

Digital signatures provide a way to verify the authenticity and integrity of messages, software, or digital documents.

```
• const crypto = require('crypto');
// Generate RSA key pair
const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa', {
modulusLength: 2048,
publicKeyEncoding: {
type: 'spki',
format: 'pem'
},
privateKeyEncoding: {
type: 'pkcs8',
format: 'pem'
}
});
// Function to sign a message
function signMessage(message, privateKey) {
const signer = crypto.createSign('sha256');
signer.update(message);
return signer.sign(privateKey, 'base64');
}
// Function to verify a signature
```

```
function verifySignature(message, signature, publicKey) {

const verifier = crypto.createVerify('sha256');

verifier.update(message);

return verifier.verify(publicKey, signature, 'base64');

}

• // Example usage

const message = 'This message needs to be signed';

const signature = signMessage(message, privateKey);

console.log('Message:', message);

console.log('Signature:', signature.substring(0, 50) + '...');

// Verify the signature

const isValid = verifySignature(message, signature, publicKey);

console.log('Signature valid:', isValid); // true

// Verify with a modified message

const isInvalid = verifySignature('Modified message', signature, publicKey);

console.log('Modified message valid:', isInvalid); // false
```

```
Message: This message needs to be signed
Signature: h/bLLKOTapO9iWl5BWyAxQuS6EhkwO...
Signature valid: true
Modified message valid: false
```

- **Random Data Generation**

• Generating secure random data is important for many cryptographic operations,

such as creating keys, salts, and initialization vectors.

• const crypto = require('crypto');

```
// Generate random bytes
const randomBytes = crypto.randomBytes(16);
console.log('Random bytes:', randomBytes.toString('hex'));
// Generate a random string (Base64)
const randomString = crypto.randomBytes(32).toString('base64');
console.log('Random string:', randomString);
// Generate a random number between 1 and 100
function secureRandomNumber(min, max) {
// Ensure we have enough randomness
const range = max - min + 1;
```

const bytesNeeded = Math.ceil(Math.log2(range) / 8);

const maxValue = 256 ** bytesNeeded;

• // Generate random bytes and convert to a number

const randomBytes = crypto.randomBytes(bytesNeeded);

const randomValue = randomBytes.reduce((acc, byte, i) => {

return acc + byte * (256 ** i);

}, 0);

// Scale to our range and shift by min

return min + Math.floor((randomValue * range) / maxValue);

}

// Example: Generate 5 random numbers

for (let i = 0; i < 5; i++) {

console.log(`Random number ${i+1}:`, secureRandomNumber(1, 100));

}

```
Random bytes: b22491a64c69879e93cd721ad55cd6a1
Random string: 1ppty6kjsQdulqzYY8xVggW5Rssx2EfX0nOMtY02EaA=
Random number 1: 95
Random number 2: 22
Random number 3: 100
Random number 4: 80
Random number 5: 45
```

Security Best Practices

• When using the Crypto module, keep these best practices in mind:

• Use modern algorithms: Avoid MD5, SHA-1, and other outdated algorithms

• Secure key management: Store keys securely, rotate them regularly, and

never hardcode them

• Use random IVs: Generate a new random IV for each encryption operation

• Add authentication: Use authenticated encryption modes like GCM when possible

• Constant-time comparisons: Always use crypto.timingSafeEqual( ) for comparing security-critical values.

• Key derivation: Use appropriate key derivation functions like scrypt, bcrypt, or PBKDF2 for password-based keys.

• Stay updated: Keep Node.js updated to get security fixes and support for newer algorithms.

• Follow standards: Adhere to established cryptographic standards and Protocols

4.7 Node.js - Callbacks Concept

What is Callback?

• A Callback in Node.js is an asynchronous equivalent for a function. It is a special type of function passed as an argument to another function.

• Node.js makes heavy use of callbacks. Callbacks help us make asynchronous calls.

• All the APIs of Node are written in such a way that they support callbacks.

• Programming instructions are executed synchronously by default. If one of the instructions in a program is expected to perform a lengthy process, the main thread of execution gets blocked.

• The subsequent instructions can be executed only after the current I/O is complete. This is where callbacks come in to the picture.

• The callback is called when the function that contains the callback as an argument completes its execution, and allows the code in the callback to run in the meantime.

• This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

The syntax of implementing callback in Node.js is as follows −

```
function function_name(argument, function (callback_argument){ // callback body })
```

• The setTimeout() function in Node.js is a typical example of callback. The following code calls the asynchronous setTimeout() method, which waits for 1000 milliseconds, but doesn't block the thread. Instead, the subsequent Hello World message, followed by the timed message.

```
setTimeout(function ( )
{
console.log('This prints after 1000 ms');
}, 1000);
console.
log("Hello World");
```

Output

```
Hello World
This prints after 1000 ms
```

• **Blocking Code Example**

• To understand the callback feature, save the following text as input.txt file.

• TutorialsPoint is the largest free online tutorials Library Master any technology. From programming languages and web development to data science and cybersecurity

• The following code reads the file synchronously with the help of readFileSync() function in fs module. Since the operation is synchronous, it blocks the execution of the rest of the code.

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
let i = 1;
while (i <=5)
{
console.log("The number is " + i);
i++;
}
```

The output shows that Node.js reads the file, displays its contents. Only after this, the following loop that prints numbers 1 to 5 is executed.

```
TutorialsPoint is the largest free online tutorials Library
Master any technology.
From programming languages and web development to data science and cybersecurity

The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
```

• We use the same input.txt file in the following code to demonstrate the use of callback.

• TutorialsPoint is the largest free online tutorials Library Master any technology. From programming languages and web development to data science and cybersecurity.

• The ReadFile() function in fs module is provided a callback function. The two arguments passed to the callback are error and the return value of ReadFile() function itself. The callback is invoked when ReadFile() finishes by returning either error or file contents. While the file read operation is inprocess, Node.js asynchronously runs the subsequent loop.

```
var fs = require("fs");
fs.readFile('input.txt', function (err, data)
{
if (err) return console.error(err);
console.log(data.toString());
});
let i = 1;
while (i <=5)
{
console.log("The number is " + i);
i++;
}
```

Output

```
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
TutorialsPoint is the largest free online tutorials Library
Master any technology.
From programming languages and web development to data science and cybersecurity
```

• Callback as Arrow function

• You can also assign an arrow function as a callback argument. Arrow function in JavaScript is an anonymous function. It is also called as lambda function. The syntax of using arrow function as Node.js callback is as follows −

• function function_name(argument, (callback_argument) => { // callback body })

• It was introduced in ES6 version of JavaScript. Let us replace the callback in the above example with an arrow function.

```
var fs = require("fs");

fs.readFile('input.txt', (err, data) =>

{

if (err) return console.error(err);

console.log(data.toString());

});

let i = 1;

while (i <=5)

{

console.log("The number is " + i);

i++;

}
```

4.8 Node.js Events

Core Concepts of Events in Node.js

Every action on a computer is an event, like when a connection is made or a file is opened.

Objects in Node.js can fire events, like the readStream object fires events when opening and closing a file:

Example

```
let fs = require('fs');

let rs = fs.createReadStream('./demofile.txt');

rs.on('open', function () {

console.log('The file is open');

});
```

```
The file is open
```

Getting Started with Events in Node.js

- Node.js uses an event-driven architecture where objects called "emitters" emit named events that cause function objects ("listeners") to be called.

- Basic Example

- // Import the events module

```
const EventEmitter = require('events');

// Create an event emitter instance

const myEmitter = new EventEmitter();
```

```
// Register an event listener
myEmitter.on('greet', () => {
console.log('Hello there!');
});
// Emit the event
myEmitter.emit('greet'); // Outputs: Hello there!
```

```
Hello there!
```

• **EventEmitter Class**

• The EventEmitter Class is fundamental to Node.js's event-driven architecture.

• It provides the ability to create and handle custom events.

• Creating an Event Emitter

• To use the EventEmitter, you need to create an instance of it:

• let events = require('events');

let eventEmitter = new events.EventEmitter();

• **The EventEmitter Object**

• You can assign event handlers to your own events with the EventEmitter object.

• In the example below we have created a function that will be executed when a "scream" event

is fired.

• To fire an event, use the emit( ) method.

EventEmitter Class

• The EventEmitter Class is fundamental to Node.js's event-driven architecture.

• It provides the ability to create and handle custom events.

• Creating an Event Emitter

• To use the EventEmitter, you need to create an instance of it:

• let events = require('events');

let eventEmitter = new events.EventEmitter();

• The EventEmitter Object

• You can assign event handlers to your own events with the EventEmitter object.

• In the example below we have created a function that will be executed when a "scream" event

is fired.

• To fire an event, use the emit( ) method.

Example

• let events = require('events');

let eventEmitter = new events.EventEmitter();

//Create an event handler:

let myEventHandler = function () {

console.log('I hear a scream!');

}

//Assign the event handler to an event:

eventEmitter.on('scream', myEventHandler);

```
//Fire the 'scream' event:

eventEmitter.emit('scream');
```

```
I hear a scream!
```

Common EventEmitter Patterns

• 1. Passing Arguments to Event Handlers

• Example

• const EventEmitter = require('events');

```
const emitter = new EventEmitter();

// Emit event with arguments

emitter.on('userJoined', (username, userId) => {

console.log(`${username} (${userId}) has joined the chat`);

});

emitter.emit('userJoined', 'JohnDoe', 42);

// Outputs: JohnDoe (42) has joined the chat
```

```
Kai (42) has joined the chat
```

2. Handling Events Only Once

• Example

• const EventEmitter = require('events');

const emitter = new EventEmitter();

// This listener will be called only once

emitter.once('connection', () => {

console.log('First connection established');

});

emitter.emit('connection'); // This will trigger the listener

emitter.emit('connection'); // This won't trigger the listener again

```
First connection established
```

3. Error Handling

• Example

• const EventEmitter = require('events');

const emitter = new EventEmitter();

// Always handle 'error' events

emitter.on('error', (err) => {

console.error('An error occurred:', err.message);

});

// This will trigger the error handler

emitter.emit('error', new Error('Something went wrong'));

```
An error occurred: Something went wrong
```

Best Practices

• 1. Always Handle Errors

• // Good practice: Always listen for 'error' events

myEmitter.on('error', (err) => {

console.error('Error in event emitter:', err);

});

• 2. Use Named Functions for Better Stack Traces

• // Instead of anonymous functions

```
function handleData(data) {
console.log('Received data:', data);
}
myEmitter.on('data', handleData);
```

• 3. Clean Up Listeners

• // Add a listener

```
const listener = () => console.log('Event occurred');
myEmitter.on('event', listener);
// Later, remove the listener when no longer needed
myEmitter.off('event', listener);
```

4.9 Node.js - Web Module

What is a Web Server?

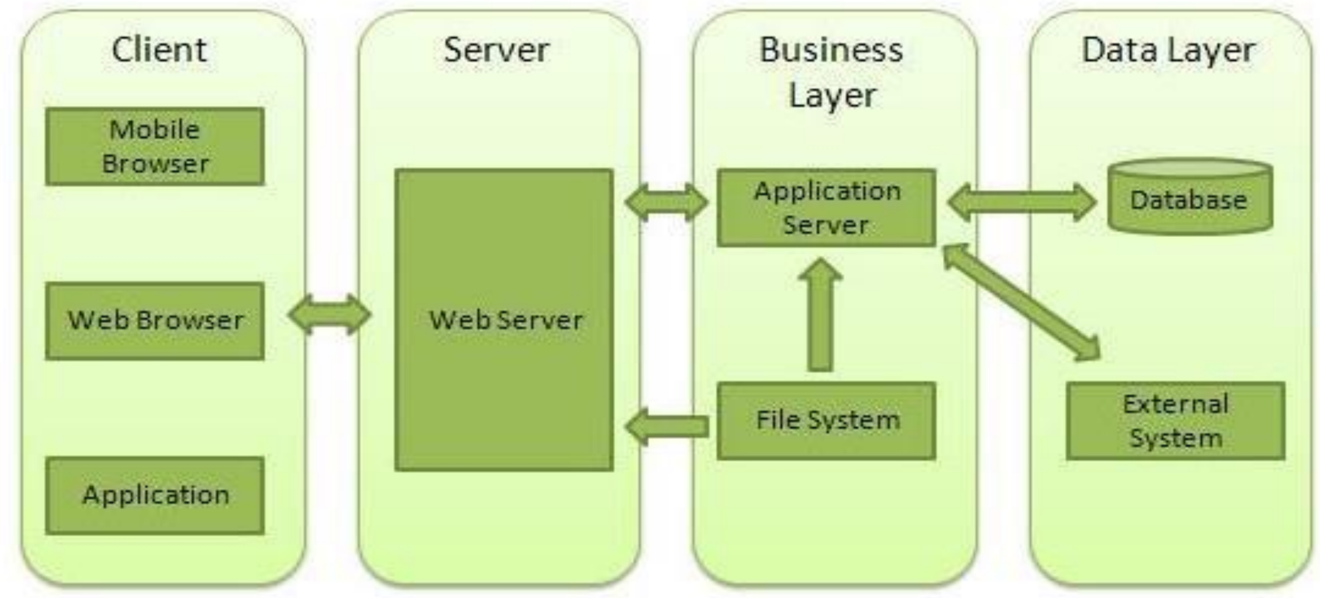
The http module in Node.js enables data transfer between the server and client over the Hyper Text Transfer Protocol (HTTP). The createServer() function in http module creates an instance of Node.js http server. It listens to the incoming requests from other http clients over the designated host and port.

The Node.js server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients. Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Web Application Architecture

A Web application is usually divided into four layers –



Client – This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.

- **Server** − This layer has the Web server which can intercept the requests made by the clients and pass them the response.
- **Business** − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.
- **Data** – This layer contains the databases or any other source of data.

Creating a Web Server using Node

Node.js provides an http module which can be used to create an HTTP client of a server. Following is the bare minimum structure of the HTTP server which listens at 5000 port.

Create a js file named server.js –

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
      } else {
         response.writeHead(200, {'Content-Type': 'text/html'});
```

```
         // Write the content of the file to response body
         console.log(data.toString());
         response.write(data.toString());
      }

      // Send the response body
      response.end();
   });
}).listen(5000);

// Console will print the message
console.log('Server running at http://127.0.0.1:5000/');
```

The createServer() function starts listening for client requests at the 5000 port of localhost. The Http Request and Server Response objects are provided internally by the Node.js server.

• It fetches the URL of HTML file requested by the HTTP client. The callback function to the createServer() function reads the requested file, and writes its contents as the servers response.

• You can send a HTTP request from a browser on the server machine itself. Run the aboce server.js file, and enter

http://localhost:5000/index.html as the URL in a browser window.

• The contents of index.html page will be rendered.

• Send a request for any other existing web page, such as http://localhost:5000/hello.html, and the requested page will be rendered.

Creating Web client using Node

A web client can be created using http module. Let's check the following example.

Create a js file named client.js (Save this file in another folder, not in the folder containing server.js script) −

```
var http = require('http');
var fs = require('fs');
var path = require('path');

// Options to be used by request
var options = {
   host: 'localhost',
   port: '5000',
   path: path.join('/',process.argv[2])
};
var body = '';
// Callback function is used to deal with response
var callback = function(response) {
   // Continuously update stream with data

   response.on('data', function(data) {
      body += data;
   });
```

```
    response.on('end', function() {
       // Data received completely.
       console.log(body);
       fs.writeFile(options.path.substr(1), body, function (err) {
         if (err)
             console.log(err);
         else
             console.log('Write operation complete.');
       });

    });
}
// Make a request to the server
var req = http.request(options, callback);
req.end();
```

This client script sends a HTTP request for a webpage given as a command-line argument. For example −

```
node main.js index.html
```

The name of the file is the argv[2] in the argument list. It is used as the path parameter in the options parameter to be passed to the http.request() method. Once the server accepts this request, its contents are written in the response stream. The client.js receives this response and writes the data received as a new file in the client's folder.

Run the server code first. From another terminal, run the client code. You will see the requested HTML file created in the client folder.

4.10 Introduction to Express.js

Express.js is a minimal and flexible Node.js web application framework that provides a list of features for building web and mobile applications easily. It simplifies the development of server-side applications by offering an easy-to-use API for routing, middleware, and HTTP utilities.

Built on Node.js for fast and scalable server-side development. Simplifies routing and middleware handling for web applications. Supports building REST APIs, real-time applications, and single-page applications. Provides a lightweight structure for flexible and efficient server-side development.

Getting Started with Express.js

Before we dive into building apps with Express.js, you need to have Node.js installed on your machine.

Follow these articles to install depending on your system:

How to Install Node.js on Windows?

Installation of Node JS on Linux

How to Install NodeJS on MacOS

How to install Express in a Node project?

First Express.js Program

• Here's how you can start with a basic Express.js application:

```
// Import Express const express= require('express');

const app=express();

// Define a route app.get('/', (req, res) =>

{

res.send('Welcome to the Express.js Tutorial');

});

// Start the server app.listen(3000, () =>

{

console.log('Server is running on http://localhost:3000');

});
```

• It will start a server, and when you visit http://localhost:3000, it will display

• Welcome to the Express.js Tutorial

• In this example:

• Express is imported using require('express'), and an app instance is created with express().

• A route is defined using the app.get() method, which responds with a message when the root URL (/) is accessed.

• The app.listen() method starts the server and listens on port 3000 for incoming requests.

• Why learn Express?

• Express.js is extremely useful because:

• It simplifies building web servers and APIs.

• Integrates seamlessly with Node.js.

• Offers extensive middleware support.

• Ideal for single-page applications and RESTful APIs.

• Express Basic

• Express.js is a minimal and flexible Node.js framework used to build web applications and APIs. It's known for its simplicity and high flexibility in handling HTTP requests.

• Introduction to Express

• Steps to create Express Application

• Design first Application using Express

• How to Structure my Application in Express JS

• Unique features of Express

• How to send response from server to client using Node and Express ?

• Why Express 'app' and 'server' files kept separately ?

• How to implement JWT authentication in Express app

• How to expire session after 1 min of inactivity in express-session of Express JS

- Express Error Handling
- **Express Functions**
- Explore the essential functions that make Express flexible and powerful. Learn how to handle various HTTP request methods and middleware.
- Express express() Function
- express.raw() Function
- express.Router() Function
- express.static() Function
- express.text() Function
- express.urlencoded() Function
- express() function Complete Reference
- **Express Applications Function**
- Understand the properties and methods of Express applications that allow configuration and response handling.
- app.locals Property
- app.mountpath Property
- Mount Event
- app.all() Function
- app.delete() Function
- app.disable() Function
- app.disabled() Function
- app.enable() Function
- app.enabled() Function

- Application Complete Reference
- **Express Requests Function**
- Get to know the request properties and methods used to handle incoming requests and extract data.
- req.app Property
- req.baseUrl Property
- req.body Property
- req.cookies Property
- req.fresh Property
- req.accepts() Function
- req.acceptsCharsets() Function
- req.acceptsEncodings() Function
- req.acceptsLanguages() Function
- Request Complete Reference
- **Express Response Function**
- Learn how to respond to HTTP requests with different status codes, cookies, and other HTTP headers.
- res.app Property
- res.headersSent Property
- res.locals Property
- res.append() Function
- res.attachment() Function
- res.cookie() Function

- res.clearCookie() Function
- res.download() Function
- res.end() Function
- Response Complete Reference
- **Express Router Function**
- Understand how to create and use routers to define reusable routing logic.
- router.all() Function
- router.METHOD() Function
- router.param() function
- router.route() Function
- router.use() Function
- Router Complete Reference
- **Express Advanced Topics**
- After learning routing and basic concepts, let's explore some advanced topics like middleware, authentication, and integrating Express with other technologies.
- Node vs Express
- HTTP request and response cycle
- Middlewares in Express
- How to update record in Cassandra using Express
- What is the use of next() function in Express JS
- How to create custom middleware in express
- Why Express is used in Web Development
- What is Express Generator

• Express HTTP methods

• How to create routes using Express and Postman?

• Why Express Is Used For Enterprise App Development

• REST API using the Express to perform CRUD

• What is express-session middleware in Express

• Key Features of Express

• Middleware and Routing: Define clear pathways (routes) within your application to handle incoming HTTP requests (GET, POST, PUT, DELETE).

• Minimalistic Design: Express.js follows a simple and minimalistic design philosophy. This simplicity allows you to quickly set up a server, define routes, and handle HTTP requests efficiently.

• Flexibility and Customization: Express.js doesn’t impose a strict application architecture. You can structure your code according to your preferences.

• Templating Power: Incorporate templating engines like Jade or EJS to generate dynamic HTML content, enhancing user experience.

• Static File Serving: Effortlessly serve static files like images, CSS, and JavaScript from a designated directory within your application.

• Node.js Integration: Express.js seamlessly integrates with the core functionalities of Node.js, allowing you to harness the power of asynchronous programming and event-driven architecture.

• Applications of Express

• Express.js empowers you to construct a wide array of web applications. Here are some captivating examples:

• RESTful APIs: Develop robust APIs that adhere to the REST architectural style, enabling communication with other applications and front-end interfaces.

• Real-time Applications: Leverage Express.js's event-driven nature to create real-time applications like chat or collaborative editing tools.

• Single-Page Applications (SPAs): Craft SPAs that fetch and update content dynamically on the client-side,offering a seamless user experience.

Express.js vs Other Frameworks

Feature	Express.js	Django	Ruby on Rails
Language	JavaScript	Python	Ruby
Flexibility	High (Unopinionated)	Moderate (Opinionated)	Low (Highly Opinionated)
Performance	High	Moderate	Moderate
Middleware Support	Extensive	Limited	Limited
Use Case	APIs, Web Apps	Full-stack Development	Full-stack Development

4.11 Express JS Installation

1. Install Node.js and npm

- Express.js runs on Node.js.
- Download and install Node.js from: https://nodejs.org ↗
- After installation, check if it's installed correctly:

```bash
node -v
npm -v
```

This will print the versions of Node.js and npm (Node Package Manager).

---

2. Create a Project Folder

Choose a location and create a new folder for your project:

```bash
mkdir my-express-app
cd my-express-app
```

---

3. Initialize the Project

Initialize a new Node.js project with `npm init`.

```bash
npm init -y
```

The `-y` flag accepts all default options and creates a `package.json` file.

4. Install Express.js

Now install Express:

```bash
npm install express
```

This will add **Express** to your `node_modules` and update `package.json`.

---

5. Create the Server File

Create a new file `index.js` (or `app.js`) and add the following code:

```js
// Load express
const express = require('express');

// Create app
const app = express();

// Define a route
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

6. Run the Server

Run the file using Node.js:

```bash
node index.js
```

If successful, you should see:

```arduino
Server running at http://localhost:3000
```

4.12 Express JS HTTP Methods

In Express.js, HTTP methods determine how a server handles client requests. They define the type of operation to perform on a resource, such as retrieving, creating, updating, or deleting data. Express allows defining routes that handle requests efficiently and in an organized manner.

Methods of HTTP Requests

Here are the different types of http requests

1. GET Method

The GET method is an HTTP request used by a client to retrieve data from the server. It takes two parameters: the URL to listen on and a callback function with req (client request) and res (server response) as arguments.

Syntax:

In the above syntax

app.get – defines a GET route in Express.

"URL" – the path the route listens to.

(req, res) => { } – callback function where req is the client request and res is the server response.

Syntax : app.get("URL",(req,res)=>{ })

• **2. POST Method**

• The POST method sends data from the client to the server, usually to store it in a database. It takes two parameters: the URL to listen on and a callback function with req (client request) and res (server response). The data sent is available in the request body and must be parsed as JSON.

• **Syntax:** app.post("URL",(req,res)=>{ })

In the above syntax

• app.post – defines a POST route in Express.

• "URL" – the path the route listens to.

• (req, res) => {} – callback function where req contains client data and res sends the server response.

Properties and Methods of Request and Response Objects in Express

• Request and Response objects both are the callback function parameters and are used for Express and Node. You can get the request query, params, body, headers, and cookies. It can overwrite any value or anything there. However, overwriting headers or cookies will not affect the output back to the browser.

• Table of Content

• Request Object

• Request Object Properties

• Request Object Methods

• Response Object

• Response Object Properties

• Response Object Method

• Request Object:

• Express uses request & response objects parameters of the callback function and are used for the Express applications. The request object represents the HTTP request and contains properties for the request query string, parameters, body, HTTP headers, etc.

• **Syntax :**

• app.get('/', function (req, res) { })

- **Request Object Properties:**
- These properties are represented below:

Request Object Methods:

Properties	Description
req.app	Used to hold a reference to the instance of the express application.
req.body	Contains key-value pairs of data submitted in the request body. By default, it is undefined and is populated when you use body-parsing middleware such as body-parser.
req.cookies	This property contains cookies sent by the request, used for the cookie-parser middleware.
req.ip	req.ip is remote IP address of the request.
req.path	req.path contains the path part of the request URL.
req.route	req.route is currently-matched route.

- There are various types of request object method, these methods are represented below: req.accepts (types). It is used to the check content types are acceptable, based on the request accept HTTP header field.

• **Example:**

• req.accepts('html');

//=>?html?

req.accepts('text/html');

// => ?text/html?

req.get(field)

• req.get(): req.get is used to returns the specified HTTP request header field.

• Example:

• req.get('Content-Type');

// => "text/plain"

req.get('content-type');

// => "text/plain"

req.get('Something');

// => undefined

req.is(type)

• If the incoming request is "CONTENT-

TYPE", this method returns true. HTTP

header field matches the MIME type by

the type parameter.

• Example: // With Content-Type: text/html;

charset=utf-8

• req.is('html');

req.is('text/html');

req.is('text/*');

// => true

• req.param(name [, defaultValue]): req.param method is used to fetch the

value of param name when present

• **Example:**

• // ?name=sonia

req.param('name')

// => "sonia"

// POST name=sonia

req.param('name')

// => "sonia"

// /user/soniafor /user/:name

req.param('name')

// => "sonia”

Response Object:

The response object specifies the HTTP response when an Express app gets an HTTP request. The response is sent back to the client browser and allows you to set new cookies value that will write to the client browser.

Response Object Properties:

Properties	Description
res.app	res.app is hold a reference to the instance of the express application that is using the middleware.
res.locals	Specify an object that contains response local variables scoped to the request.

Response Object Method:

There are various types of response object method, these methods are represented below :

Response Append Method:

Syntax:

```
res.append(field, [value])
```

Response append method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate so this method redress that.

Example :

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
res.append('Warning', '299 Miscellaneous warning');
```

Response Attachment Method:

Syntax:

```
res.attachment('path/to/js_pic.png');
```

Response attachment method allows you to send a file as an attachment in the HTTP response.

Example:

```
res.attachment('path/to/js_pic.png');
```

Response Cookie Method:

Syntax:

```
res.cookie(name, value [, options])
```

It is used to set a cookie name to value. The value can be a string or object converted to JSON.

Example:

```
res.cookie('name', 'alish', { domain: '.google.com', path: '/admin', secure: true });
res.cookie('Section', { Names: [sonica,riya,ronak] });
res.cookie('Cart', { items: [1,2,3] }, { maxAge: 900000 });
Response Download Method
Syntax:   res.download(path [, filename] [, fn])
```

Example:

```
res.download('/report-12345.pdf');
```

res.download method is transfer file at path as an "attachment" and the browser to prompt user for download.

Response End Method:

Syntax:

```
res.end([data] [, encoding])
```

Response end method is used to end the response process.

Example:

```
res.end();
res.status(404).end();
Response Get Method
Syntax :   res.get(field)
```

res.get method provides HTTP response header specified by field.

Example:

```
res.get('Content-Type');
Response JSON Method
Syntax :   res.json([body])
```

Response JSON method returns the response in JSON format.

Example :

```
res.json(null)
res.json({ name: 'alish' })
Response Render Method
Syntax :   res.render(view [, locals] [, callback])
```

Response render method renders a view and sends the rendered HTML string to the client.

Example :

```
// send the rendered view to the client
res.render('index');
// pass a local variable to the view
res.render('user', { name: 'monika' }, function(err, html) {
 // ...
});
```

Response Status Method:

Syntax:

```
res.status(code)
```

res.status method sets an HTTP status for the response.

Example: res.status(403).end():

```
res.status(400).send('Bad Request');
```

Response Type Method:

Syntax:

```
res.type(type)
```

res.type method sets the content-type HTTP header to the MIME type.

Example:

```
res.type('.html');              // => 'text/html'
res.type('html');               // => 'text/html'
res.type('json');               // => 'application/json'
res.type('application/json');   // => 'application/json'
res.type('png');                // => image/png:
```

4.13 ExpressJS – Cookies

Cookies are simple, small files/data that are sent to client with a server request and stored on the client side. Every time the user loads the website back, this cookie is sent with the request.

This helps us keep track of the users actions.

The following are the numerous uses of the HTTP Cookies −

- Session management
- Personalization(Recommendation systems)
- User tracking

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code –

To use cookies with Express, we need the cookie-parser middleware. To install it, use the following code –

```
E:\Dev\hello-world>npm install --save cookie-parser
up to date, audited 128 packages in 2s

20 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Now to use cookies with Express, we will require the **cookie-parser**. cookie-parser is a middleware which parses cookies attached to the client request object. To use it, we will require it in our **index.js** file; this can be used the same way as we use other middleware. Here, we will use the following code.

```
var cookieParser = require('cookie-parser');
app.use(cookieParser());
```

cookie-parser parses Cookie header and populates **req.cookies** with an object keyed by the cookie names. To set a new cookie, let us define a new route in your Express app like –

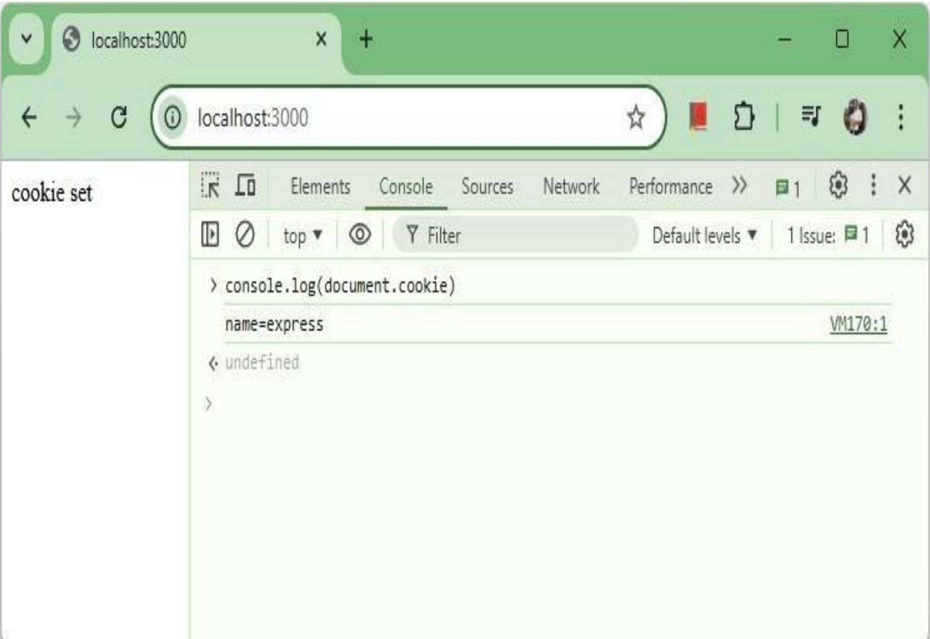
index.js

```
var express = require('express');
var app = express();

var cookieParser = require('cookie-parser');
app.use(cookieParser());

app.get('/', function(req, res){
   res.cookie('name', 'express').send('cookie set'); //Sets name = express
});

app.listen(3000);
```

To check if your cookie is set or not, just go to your browser, fire up the console, and enter –

```
console.log(document.cookie);
```

You will get the output like (you may have more cookies set maybe due to extensions in your browser) –



The browser also sends back cookies every time it queries the server. To view cookies from your server, on the server console in a route, add the following code to that route.

```
console.log('Cookies: ', req.cookies);
```

Next time you send a request to this route, you will receive the following output.

```
Cookies: { name: 'express' }
```

Adding Cookies with Expiration Time

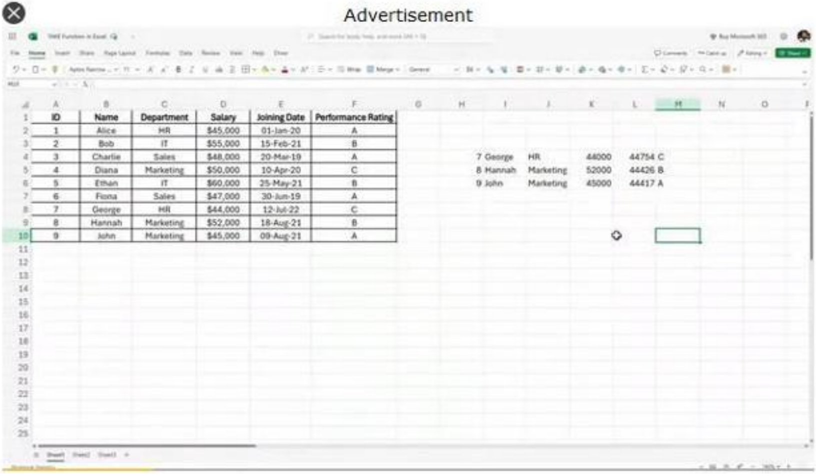
You can add cookies that expire. To add a cookie that expires, just pass an object with property 'expire' set to the time when you want it to expire. For example,

```
//Expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {expire: 360000 + Date.now()});
```

Another way to set expiration time is using **'maxAge'** property. Using this property, we can provide relative time instead of absolute time. Following is an example of this method.

```
//This cookie also expires after 360000 ms from the time it is set.
res.cookie(name, 'value', {maxAge: 360000});
```

Advertisement



Deleting Existing Cookies

To delete a cookie, use the clearCookie function. For example, if you need to clear a cookie named **foo**, use the following code.

index.js

```
var express = require('express');
var app = express();

app.get('/clear_cookie_foo', function(req, res){
   res.clearCookie('foo');
   res.send('cookie foo cleared');
});

app.listen(3000);
```

4.14 Middleware Routing

Express.js middleware routing involves using middleware functions to process incoming requests before they reach the final route handler. Middleware functions have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.

Key Concepts:

• Middleware Functions:

These are functions that execute between the client sending a request and the server sending a response. They can perform various tasks like:

• Modifying req and res objects.

• Performing authentication or authorization checks.

• Logging requests.

• Handling errors.

• Ending the request-response cycle.

• Calling next( ) to pass control to the next middleware or route handler.

• app.use() : This method registers middleware functions at the application level. When called without a path, the middleware applies to all incoming requests.

When called with a path, the middleware applies only to requests matching that path or its sub-paths.

- **Route-Specific Middleware:** Middleware can be applied to specific routes or groups of routes using `app.get()`, `app.post()`, etc., or by using `express.Router()`.
    - **Directly in route handlers:**

JavaScript

```
app.get('/admin', authMiddleware, (req, res) => {
  res.send('Admin dashboard');
});
```

- Using `express.Router()`: This allows modular organization of routes and their associated middleware.

JavaScript

```
const express = require('express');
const router = express.Router();

router.use(authMiddleware); // This middleware applies to all rou

router.get('/profile', (req, res) => {
  res.send('User profile');
});

module.exports = router;
```

Order of Execution: Middleware functions are executed in the order they are defined. If a middleware function does not end the request-response cycle (e.g., by sending a response), it must call next() to pass control to the subsequent middleware or route handler.

Example of Middleware Routing:

JavaScript

```javascript
const express = require('express');
const app = express();

// Global middleware for logging requests
app.use((req, res, next) => {
  console.log(`Request received: ${req.method} ${req.url}`);
  next(); // Pass control to the next middleware or route handler
});

// Middleware for authentication (applied to specific routes)
const authMiddleware = (req, res, next) => {
  if (req.headers.authorization === 'Bearer mysecrettoken') {
    next();
  } else {
    res.status(401).send('Unauthorized');
  }
};

// Route without authentication
app.get('/', (req, res) => {
  res.send('Welcome to the homepage!');
});

// Route with authentication middleware
app.get('/protected', authMiddleware, (req, res) => {
  res.send('This is a protected route!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

Using middleware

• Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

• Middleware functions are functions that have access to the request object(req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

• Middleware functions can perform the following tasks:

• Execute any code.

• Make changes to the request and the response objects.

• End the request-response cycle.

• Call the next middleware function in the stack.

• If the current middleware function does not end the request-response cycle, it must call next( ) to pass control to the next middleware function. Otherwise, the request will be left hanging.

• An Express application can use the following types of middleware:

• Application-level middleware

• Router-level middleware

• Error-handling middleware

• Built-in middleware

• Third-party middleware

• You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system.

Application-level middleware

Bind application-level middleware to an instance of the app object by using the `app.use()` and `app.METHOD()` functions, where `METHOD` is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.

This example shows a middleware function with no mount path. The function is executed every time the app receives a request.

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})
```

This example shows a middleware function mounted on the `/user/:id` path. The function is executed for any type of HTTP request on the `/user/:id` path.

```
app.use('/user/:id', (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

This example shows a route and its handler function (middleware system). The function handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  res.send('USER')
})
```

Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the `/user/:id` path.

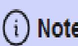
```
app.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the `/user/:id` path. The second route will not cause any problems, but it will never get called because the first route ends the request-response cycle.

This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  console.log('ID:', req.params.id)
  next()
}, (req, res, next) => {
  res.send('User Info')
})

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

To skip the rest of the middleware functions from a router middleware stack, call `next('route')` to pass control to the next route.



Note

`next('route')` will work only in middleware functions that were loaded by using the `app.METHOD()` or `router.METHOD()` functions.

This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
app.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
  else next()
}, (req, res, next) => {
  // send a regular response
  res.send('regular')
})

// handler for the /user/:id path, which sends a special response
app.get('/user/:id', (req, res, next) => {
  res.send('special')
})
```

Middleware can also be declared in an array for reusability.

This example shows an array with a middleware sub-stack that handles GET requests to the `/user/:id` path

```
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod (req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

const logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, (req, res, next) => {
  res.send('User Info')
})
```

Router-level middleware

Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of `express.Router()`.

```
const router = express.Router()
```

Load router-level middleware by using the `router.use()` and `router.METHOD()` functions.

The following example code replicates the middleware system that is shown above for application-level middleware, by using router-level middleware:

```
const express = require('express')
const app = express()
const router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router
router.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
router.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})

// a middleware sub-stack that handles GET requests to the /user/:id path
router.get('/user/:id', (req, res, next) => {
  // if the user ID is 0, skip to the next router
  if (req.params.id === '0') next('route')
  // otherwise pass control to the next middleware function in this stack
  else next()
}, (req, res, next) => {
  // render a regular page
  res.render('regular')
})
```

```
// handler for the /user/:id path, which renders a special page
router.get('/user/:id', (req, res, next) => {
  console.log(req.params.id)
  res.render('special')
})

// mount the router on the app
app.use('/', router)
```

To skip the rest of the router's middleware functions, call `next('router')` to pass control back out of the router instance.

This example shows a middleware sub-stack that handles GET requests to the `/user/:id` path.

```
const express = require('express')
const app = express()
const router = express.Router()

// predicate the router with a check and bail out when needed
router.use((req, res, next) => {
  if (!req.headers['x-auth']) return next('router')
  next()
})

router.get('/user/:id', (req, res) => {
  res.send('hello, user!')
})

// use the router and 401 anything falling through
app.use('/admin', router, (req, res) => {
  res.sendStatus(401)
})
```

Error-handling middleware

> Error-handling middleware always takes ***four*** arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the `next` object, you must specify it to maintain the signature. Otherwise, the `next` object will be interpreted as regular middleware and will fail to handle errors.

Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature `(err, req, res, next)`:

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

For details about error-handling middleware, see: Error handling.

Built-in middleware

Starting with version 4.x, Express no longer depends on Connect. The middleware functions that were previously included with Express are now in separate modules; see the list of middleware functions.

Express has the following built-in middleware functions:

- express.static serves static assets such as HTML files, images, and so on.
- express.json parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- express.urlencoded parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

Third-party middleware

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function `cookie-parser`.

```
$ npm install cookie-parser
```

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

UNIT-V

Introduction to Mongo DB, Differences between SQL and NOSQL, Mongo DB datatypes, Mongo DB installation, Data Modeling in Mongo DB- Create Database, Insert Mongo DB, Update Mongo DB,find Mongo DB , Delete Mongo Db, Queries in Mongo DB.

MongoDB is a popular, open-source NoSQL database management system, often categorized as a document-oriented database. It differs significantly from traditional relational databases (like SQL databases) in its data storage and retrieval methods.

Key Concepts of MongoDB:

Document-Oriented:

Instead of storing data in tables with rows and columns, MongoDB stores data in flexible, JSON-like documents. These documents are stored in a binary representation known as BSON (Binary JSON).

Dynamic Schema:

MongoDB's documents do not require a predefined schema, offering flexibility in how data is structured. Documents within the same collection can have different fields and structures, adapting easily to evolving application needs.

Collections:

Documents are organized into collections, which are analogous to tables in relational databases. A collection is a group of related documents within a single database

Scalability:

MongoDB is designed for horizontal scalability, allowing users to distribute data across multiple servers (sharding) to handle large datasets and high traffic loads.

NoSQL:

As a NoSQL database, MongoDB offers an alternative to the rigid structure of relational databases, particularly suited for handling unstructured and semi-structured data.

Features:

MongoDB includes features such as ad-hoc queries, indexing for efficient data retrieval, replication for high availability, and aggregation frameworks for data processing.

- **Benefits of using MongoDB:**
- **Flexibility:**
- The dynamic schema allows for rapid development and easy adaptation to changing data requirements.
- **Scalability:**
- Its distributed architecture enables horizontal scaling to accommodate growing data volumes and user demands.
- **Performance:**
- Optimized for high-volume data storage and quick retrieval, especially with its indexing capabilities.
- **Ease of Use:**
- MongoDB offers intuitive APIs and tools, including the MongoDB Shell and GUI tools like MongoDB Compass, simplifying database interaction and management.

5.1 MongoDB Introduction

- MongoDB is a popular document-oriented NoSQL database that stores data in a flexible BSON format, enabling fast and efficient storage and retrieval. It is available under the SSPL, which is not recognized as an open-source license by the Open Source Initiative due to commercial use restrictions.

01
Document-Oriented NoSQL Database

02
Stores Data in Flexible BSON/ JSON-like Format

03
Scales Horizontaly

04
Handles Unstrurtured & Semi-Data

05
Open-Source & Cross-Platform

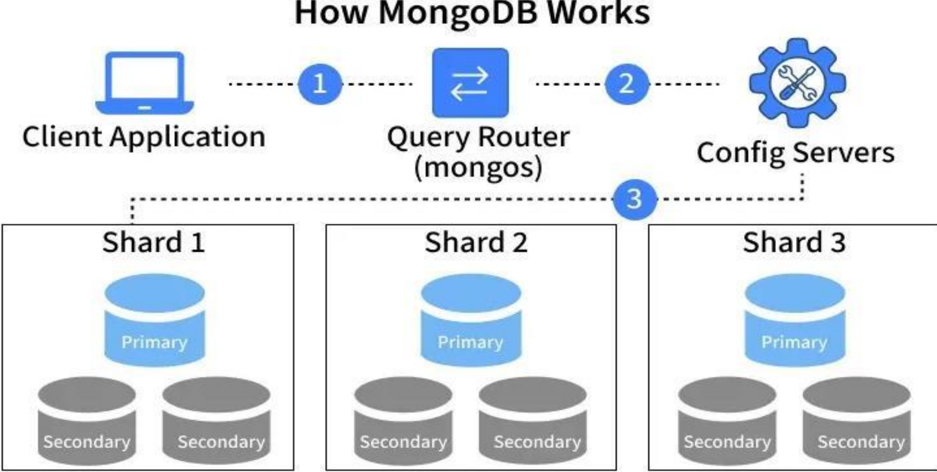
- **A simple MongoDB document Structure:**

```
{
"_id": 1,
"title": "Geeksforgeeks",
"author": "John Smith",
"url": "https://www.geeksforgeeks.org/",
"tags": ["NoSQL", "Database", "Tutorial"],
"published": true,
"views": 1500
}
```

- **In the above structure:**
- **_id:** Unique identifier for the document.
- **title, author, url:** Basic fields storing strings.
- **tags:** An array of strings.
- **published:** A Boolean value.
- **views:** A number representing views or count.

How MongoDB Works

In a MongoDB sharded cluster, the components work together as follows:



- **Client Application:** The application sends requests to MongoDB to read or write data.
- **Query Router (mongos):** The query router receives the client request and determines which shard should handle it.
- **Config Servers:** Store metadata about the cluster, including which data resides on which shard.
- **Shards:** Data is horizontally distributed across replica sets, with the primary handling writes and secondaries providing replication and reads.
- **Primary Node:** Handles all write operations.
- **Secondary Nodes:** Replicate data from the primary for high availability and read operations.

Basic SQL Operations

MongoDB provides several basic operations to manage and manipulate data efficiently:

- **Create (Insert):** Add new documents to a collection.
- **Read (Find):** Retrieve documents from a collection based on query criteria.
- **Update:** Modify existing documents in a collection.
- **Delete:** Remove documents from a collection.

Features of the MongoDB database

MongoDB offers several key features that make it efficient and scalable:

- **Document Oriented:** Stores all related data together in a single, flexible document rather than multiple tables.
- **Indexing:** Enables fast queries by avoiding full collection scans, crucial for handling large data efficiently.
- **Scalability:** Uses sharding to distribute data across multiple servers; easily add new machines to expand.
- **Replication and High Availability:** Stores multiple data copies on different servers for redundancy and fault tolerance.
- **Aggregation:** Processes and summarizes data (like SQL GROUP BY) using operations such as sum, avg, min, and max.

Real-World Applications of MongoDB

MongoDB powers a wide range of modern applications:

- **Websites & Apps:** Platforms like e-commerce sites use MongoDB to store product catalogs, user profiles, and order histories.
- **Real-Time Analytics:** Companies analyze user activity or sales trends on the fly, such as tracking popular products during a sale.
- **Content Management & Personalization:** Apps manage dynamic content and personalize recommendations, like suggesting movies based on viewing history.
- **Big Data & IoT:** MongoDB handles large volumes of unstructured data from sensors, logs, or social media feeds for processing and insights.

- **Language Support by MongoDB**
- MongoDB offers official drivers for many popular programming languages, enabling easy integration and database operations.

- C & C++:
- Rust
- C#
- Java
- Node.js
- Perl & PHP
- Python
- Ruby & Scala
- Go
- Erlang

5.2 Difference between SQL and NoSQL

Aspect	SQL (Relational)	NoSQL (Non-relational)
Data Structure	Tables with rows and columns	Document-based, key-value, column-family, or graph-based
Schema	Fixed schema (predefined structure)	Flexible schema (dynamic and adaptable)
Scalability	Vertically scalable (upgrading hardware)	Horizontally scalable (adding more servers)
Data Integrity	ACID-compliant (strong consistency)	BASE-compliant (more available, less consistent)
Query Language	SQL (Structured Query Language)	Varies (e.g., MongoDB uses its own query language)
Performance	Efficient for complex queries and transactions	Better for large-scale data and fast read/write operations
Use Case	Best for transactional systems (banking, ERP, etc.)	Ideal for big data, real-time web apps, and data lakes
Examples	MySQL, PostgreSQL, Oracle, MS SQL Server	MongoDB, Cassandra, CouchDB, Neo4j

1. Type

- SQL databases are primarily called Relational Databases (RDBMS).
- whereas NoSQL databases are primarily called non-relational or distributed databases.

2. Language

- SQL databases define and manipulate data-based structured query language (SQL). Seeing from a side this language is extremely powerful. SQL is one of the most **versatile** and **widely-used options** available which makes it a **safe choice**, especially for great **complex queries**. But from another side, it can be restrictive.
- SQL requires you to use predefined schemas to determine the structure of your data before you work with it. Also, all of our data must follow the same structure. This can require significant **up-front preparation** which means that a change in the structure would be both difficult and disruptive to your whole system.

3. Scalability

- In almost all situations SQL databases are vertically scalable. This means that you can increase the load on a single server by increasing things like RAM, CPU, or SSD. But on the other hand, NoSQL databases are **horizontally scalable**. This means that you handle more traffic by **sharing**, or adding more servers in your **NoSQL database**.
- It is similar to adding more floors to the same building versus **adding more buildings** to the neighborhood. Thus NoSQL can ultimately become larger and more powerful, making these databases the preferred choice for large or ever-changing data sets.
- **4. Structure**
- **SQL databases** are table-based on the other hand **NoSQL databases** are either **key-value pairs**, **document-based**, **graph databases**, or **wide-column stores**. This makes relational SQL databases a better option for applications that require **multi-row transactions** such as an accounting system or for legacy systems that were built for a relational structure.

- Here is a simple example of how a structured data with rows and columns vs a non-structured data without definition might look like. A product table in SQL **db** might accept data looking like this:
- {
  "id": "101",
  "category":"food"
  "name":"Apples",
  "qty":"150"
  }
- Whereas a unstructured NOSQL DB might save the products in many variations without constraints to change the underlying table structure

  Products=[

  {

  "id":"101:",

  category":"food",

  "name":"California Apples"

  ,"qty":"150"},

  {"id":"102,

  "category":"electronics"

  "name":"Apple MacBook Air",

  "qty":"10",

  "specifications":{ "storage":"256GB SSD",

  "cpu":"8 Core",

  "camera": "1080p FaceTime HD camera" }

  }

]

5. Property followed

- SQL databases follow ACID properties (Atomicity, Consistency, Isolation, and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability, and Partition tolerance).

6. Support

- Great support is available for all **SQL databases** from their vendors. Also, a lot of independent consultants are there who can help you with SQL databases for very large-scale deployments but for some **NoSQL databases** you still have to rely on community support and only limited outside experts are available for setting up and deploying your **large-scale NoSQL deploy**.
- **Function of SQL**
- SQL databases, also known as **Relational Database Management Systems (RDBMS)**, use structured tables to store data. They rely on a **predefined schema** that determines the organization of data within tables, making them suitable for applications that require a fixed, consistent structure.
- **Structured Data**: Data is organized in tables with rows and columns, making it easy to relate different types of information.
- **ACID Compliance**: SQL databases follow the ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure reliable transactions and data integrity.
- **Examples**: Popular SQL databases include **MySQL**, **PostgreSQL**, **Oracle**, and **MS SQL Server**.
- **Function Of NoSql**
- NoSQL databases, on the other hand, are designed to handle **unstructured or semi-structured data**. Unlike SQL databases, NoSQL offers **dynamic schemas** that allow for more flexible data storage, making them ideal for handling massive volumes of data from various sources.

- **Flexible Schema**: NoSQL databases allow the storage of data without a predefined structure, making them more adaptable to changing data requirements.
- **CAP Theorem**: NoSQL databases are designed based on the **CAP theorem** (Consistency, Availability, Partition Tolerance), which prioritizes availability and partition tolerance over strict consistency.
- **Examples**: Well-known NoSQL databases include **MongoDB**, **Cassandra**, **CouchDB**, and **HBase**.
- **SQL vs NoSQL: Which is Faster?**
- **SQL Databases**: Generally, SQL databases perform well for **complex queries**, structured data, and systems requiring **data consistency** and **integrity**. However, as the volume of data grows, they may struggle with **scalability** and may require significant infrastructure upgrades.
- **NoSQL Databases**: NoSQL databases excel in scenarios that demand **high performance** and **scalability**. Because of their **horizontal scalability** (accommodating more servers), they handle large amounts of data and high-velocity workloads better. For instance, MongoDB or Cassandra is a common choice when dealing with big data or applications with high traffic
- **When to Choose SQL?**
- SQL databases are well-suited for use cases where:
- **Data consistency** and **transactional integrity** are critical (e.g., banking systems, customer relationship management).
- The application needs a **well-defined schema** and structured data.
- Complex queries and **relational data** are involved.
- Applications requiring **multi-row transactions** (such as inventory management) benefit from SQL's robust features.

- **When to Choose NoSQL?**
- NoSQL databases are a better choice when:
- You need to handle **large, unstructured data** sets, like social media data or logs.
- The application requires **horizontal scalability** to accommodate high traffic and big data.
- There is a need for **real-time data processing** and **flexible data models** (e.g., a content management system).
- You are dealing with applications requiring **frequent changes in data structures**.

5.3 DataTypes in MongoDB

- In MongoDB, data types define the kind of data a field can hold, such as text, numbers, dates, or binary data. Choosing the right data type is important because it determines how data is stored, retrieved, and used in queries. Understanding MongoDB data types helps you organize your data efficiently and work with it effectively.
- **Below are the most commonly used MongoDB data types.**

1. String

It is one of the most commonly used data types in MongoDB, as it stores the data in BSON string format. The MongoDB driver automatically handles the conversion between the programming language string type and BSON by encoding and decoding strings in UTF-8 during serialization and deserialization. All strings stores in MongoDB must be utf-8.

Example: In the following example we are storing the name of the student in the student collection:

```
Command Prompt - mongo
switched to db gfg
> db.student.insertMany([{name:"Akshay"},{name:"Vikash"}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("601af2dd6fd54aa34c9c6df3"),
                ObjectId("601af2dd6fd54aa34c9c6df4")
        ]
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
>
```

Here, the `name` field is a string representing the student's name.

Key Features:

- Stored in UTF-8 format.
- Ideal for storing text-based data.

2. Integer

In MongoDB, the integer data type is used to store an integer value. We can store integer data type in two forms 32-bit signed integer and 64-bit signed integer. These are used to store whole numbers, such as ages, counts, or any other numerical data that doesn't require decimal points.

Example: In the following example we are storing the age of the student in the student collection:

```
> db.student.insertOne({name:"Akash",age:19})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af3456fd54aa34c9c6df5")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
>
```

In this example, `age` is stored as an integer.

Key Features:

- It Can be represented as either a 32-bit or 64-bit signed integer.
- It is Suitable for storing whole numbers.

3. Double

The double data type is used for storing floating-point numbers (decimal values). It's commonly used for storing data that requires decimal precision, such as prices, percentages, or scores.

Example: In the following example we are storing the marks of the student in the student collection:

```
}
> db.student.insertOne({name:"Sagar",age:20,marks:546.43})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af4126fd54aa34c9c6df6")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
>
```

In this example, `marks` is stored as a double to store the decimal value.

Key Features:

- It is used to store floating-point numbers.
- It is Ideal for data requiring decimal precision.

4. Boolean

The boolean data type stores one of two values: true or false. It's used for representing binary states, such as "active/inactive" or "pass/fail."

Example: In the following example we are storing the final result of the student as pass or fail in boolean values.

```
> db.student.insertOne({name:"vishal",pass:true})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af47d6fd54aa34c9c6df7")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
>
```

In this example, the `passed` field is a boolean value indicating whether the student passed or failed.

Key Features:

- It Stores `true` or `false` values.
- It is Commonly used for binary states.

5. Null

The **null** data type stores a **null** value. This is useful when you want to represent the absence of data, such as an optional field that may not be set.

Example: In the following example, the student does not have a mobile number so the number field contains the value null.

```
> db.student.insertOne({name:"Akash",MobNo:null,skills:["c","c++","java","python","JS"]})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af5fb6fd54aa34c9c6df8")
}
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
{
        "_id" : ObjectId("601af5fb6fd54aa34c9c6df8"),
        "name" : "Akash",
        "MobNo" : null,
        "skills" : [
                "c",
                "c++",
                "java",
                "python",
                "JS"
        ]
}
>
```

Here, `phone_number` is set to `null`, indicating that the student has not provided a phone number.

Key Features:

- It Represents the absence of data.
- It is used for missing or undefined values.

6. Array

The array data type allows us to store multiple values in a single field.MongoDB arrays can contain values of the same or different data types, providing flexibility in how you store collections of related data. In MongoDB, the array is created using square brackets([]).

Example: In the following example, we are storing the technical skills of the student as an array.

```
> db.student.find().pretty()
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df3"), "name" : "Akshay" }
{ "_id" : ObjectId("601af2dd6fd54aa34c9c6df4"), "name" : "Vikash" }
{
        "_id" : ObjectId("601af3456fd54aa34c9c6df5"),
        "name" : "Akash",
        "age" : 19
}
{
        "_id" : ObjectId("601af4126fd54aa34c9c6df6"),
        "name" : "Sagar",
        "age" : 20,
        "marks" : 546.43
}
{
        "_id" : ObjectId("601af47d6fd54aa34c9c6df7"),
        "name" : "vishal",
        "pass" : true
}
{
        "_id" : ObjectId("601af5fb6fd54aa34c9c6df8"),
        "name" : "Akash",
        "MobNo" : null,
        "skills" : [
                "c",
                "c++",
                "java",
                "python",
                "JS"
        ]
}
>
```

In this example, the `skills` field stores an array of strings representing the student's technical skills.

Key Features:

- It is used to Store multiple values in a single field.
- It can hold values of different data types.

7. Object (Embedded Document)

Object data type stores embedded documents. Embedded documents are also known as nested documents. Embedded document or nested documents are those types of documents which contain a document inside another document. Embedded documents allow us to structure our data hierarchically, which is useful for representing more complex data models.

Example: In the following example, we are storing all the information about a book in an embedded document.

```
> db.book.insertOne({Book:{name:"C in depth",writer:"Aaksh"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af71f6fd54aa34c9c6df9")
}
> db.bool.find().pretty()
> db.book.find().pretty()
{
        "_id" : ObjectId("601af71f6fd54aa34c9c6df9"),
        "Book" : {
                "name" : "C in depth",
                "writer" : "Aaksh"
        }
}
>
```

Here, the `book` field stores an embedded document with properties like `title`, `author`, and `published`.

Key Features:

- It is used to Store nested documents.
- It is useful for hierarchical data structures.

8. Object Id

Whenever we create a new document in the collection MongoDB automatically creates a unique object id for that document(if the document does not have it). There is an _id field in MongoDB for each document. The data which is stored in Id is of hexadecimal format and the length of the id is 12 bytes which consist:

- 4-bytes for Timestamp value.
- 5-bytes for Random values. i.e., 3-bytes for machine Id and 2-bytes for process Id.
- 3- bytes for Counter

We can also create your own id field, but make sure that the value of that id field must be unique.

Example: In the following example, when we insert a new document it creates a new unique object id for it.

```
> db.book.insertOne({Book:{name:"C in depth",writer:"Aaksh"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af71f6fd54aa34c9c6df9")
}
> db.bool.find().pretty()
> db.book.find().pretty()
{
        "_id" : ObjectId("601af71f6fd54aa34c9c6df9"),
        "Book" : {
                "name" : "C in depth",
                "writer" : "Aaksh"
        }
}
>
```

In this example, MongoDB automatically generates a unique `_id` for each document.

Key Features:

- Unique identifier for each document.
- Automatically generated by MongoDB.

9. Undefined

The undefined data type represents a value that is not defined. It is rarely used in modern MongoDB applications, as it has been replaced by the `null` value for most practical purposes.

Example: In the following example the type of the duration of the project is undefined.

```
        }
}
> db.project.insertOne({name:"FauG",duration:undefined,binaryValue:"110011001"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af8516fd54aa34c9c6dfa")
}
> db.project.find().pretty()
{
        "_id" : ObjectId("601af8516fd54aa34c9c6dfa"),
        "name" : "FauG",
        "duration" : undefined,
        "binaryValue" : "110011001"
}
>
```

In this example, the `duration` field is set to `undefined`.

Key Features:

- It is used to represents an undefined value.
- It is typically replaced by `null value`.

10. Binary Data

The binary data data type stores binary information such as images, files, or encrypted data. Binary data is often used when working with non-textual data.

Example: In the following example the value stored in the binaryValue field is of binary type.

```
> db.project.insertOne({name:"FauG",duration:undefined,binaryValue:"110011001"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af8516fd54aa34c9c6dfa")
}
> db.project.find().pretty()
{
        "_id" : ObjectId("601af8516fd54aa34c9c6dfa"),
        "name" : "FauG",
        "duration" : undefined,
        "binaryValue" : "110011001"
}
>
```

Here, the `image` field stores binary data.

Key Features:

- It is used for storing non-textual data (e.g., images, files).
- It is stored as binary data.

11. Date

Date data type stores date. It is a 64-bit integer which represents the number of milliseconds. BSON data type generally supports UTC datetime and it is signed. If the value of the date data type is negative then it represents the dates before 1970. There are various methods to return date, it can be returned either as a string or as a date object. Some method for the date:

- **Date():** It returns the current date in string format.
- **new Date(): It** returns a date object. It uses the ISODate() wrapper.
- **new ISODate():** It also returns a date object. It uses the ISODate() wrapper.

Example: In the following example we are using all the above method of the date:

```
>
> var date1  = Date()
> var date2 = new Date()
> var date3 = new ISODate()
> db.date.insertOne({Date1:date1,Date2:date2,Date3:date3})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afa326fd54aa34c9c6dfc")
}
> db.date.find().pretty()
{
        "_id" : ObjectId("601afa326fd54aa34c9c6dfc"),
        "Date1" : "Thu Feb 04 2021 01:00:34 GMT+0530 (India Standard Time)",
        "Date2" : ISODate("2021-02-03T19:30:40.014Z"),
        "Date3" : ISODate("2021-02-03T19:30:56.454Z")
}
>
```

In this example, the `created_at` field stores the current date and time.

Key Features:

- It is used to store dates and times as 64-bit integers.
- It is commonly used for timestamps.

12. Min & Max key

The MinKey and MaxKey data types are used for internal comparisons. MinKey represents the lowest BSON element, while MaxKey represents the highest.

Example:

```
>
> db.m.insertOne({min:MinKey,max:MaxKey})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afbd66fd54aa34c9c6dff")
}
> db.m.find().pretty()
{
        "_id" : ObjectId("601afbd66fd54aa34c9c6dff"),
        "min" : { "$minKey" : 1 },
        "max" : { "$maxKey" : 1 }
}
>
```

Key Features:

- It is used for internal comparisons in queries.
- It is used to represent the lowest and highest BSON elements.

13. Symbol

This data type similar to the string data type. It is generally not supported by a mongo shell, but if the shell gets a symbol from the database, then it converts this type into a string type.

Example:

```
}
> var symbl = "nik432#"
> db.symbl.insert({symbol:symbl})
WriteResult({ "nInserted" : 1 })
> db.symbl.find().pretty()
{ "_id" : ObjectId("601afb6f6fd54aa34c9c6dfe"), "symbol" : "nik432#" }
>
```

14. Regular Expression

The regular expression data type is used to store regex patterns. MongoDB supports regex queries for performing pattern matching on string fields.

Example: In the following example we are storing the regular expression gfg:

```
        binaryvalue  :  110011001
}
> var expression = new RegExp("%gfg")
> db.exp.insertOne({Regular Expression: expression})
uncaught exception: SyntaxError: missing : after property id :
@(shell):1:26
> db.exp.insertOne({RegularExpression: expression})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af91f6fd54aa34c9c6dfb")
}
> db.exp.find().pretty()
{
        "_id" : ObjectId("601af91f6fd54aa34c9c6dfb"),
        "RegularExpression" : /%gfg/
}
>
```

In this example, the `name` field stores a regular expression to match names that contain "John" case-insensitively.

Key Features:

- It is used for pattern matching with regular expressions.
- It is useful for querying string data.

15. JavaScript

MongoDB allows us to store JavaScript code within documents using the JavaScript data type. This can be useful for storing code or expressions.

Example: In this example, we are using the JavaScript syntax in the shell:

```
        binaryValue" : "110011001"
}
> var expression = new RegExp("%gfg")
> db.exp.insertOne({Regular Expression: expression})
uncaught exception: SyntaxError: missing : after property id :
@(shell):1:26
> db.exp.insertOne({RegularExpression: expression})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601af91f6fd54aa34c9c6dfb")
}
> db.exp.find().pretty()
{
        "_id" : ObjectId("601af91f6fd54aa34c9c6dfb"),
        "RegularExpression" : /%gfg/
}
>
```

Key Features:

- It is used to store JavaScript code.
- It is useful for embedded code or expressions.

16. JavaScript with Scope

The JavaScript with Scope data type in MongoDB was used to store JavaScript code along with its scope (variables and functions). This feature has been deprecated in MongoDB 4.4 and is no longer recommended for use.

Example: In this example, we are using the JavaScript syntax in the shell:

```
>
> var date1  = Date()
> var date2 = new Date()
> var date3 = new ISODate()
> db.date.insertOne({Date1:date1,Date2:date2,Date3:date3})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afa326fd54aa34c9c6dfc")
}
> db.date.find().pretty()
{
        "_id" : ObjectId("601afa326fd54aa34c9c6dfc"),
        "Date1" : "Thu Feb 04 2021 01:00:34 GMT+0530 (India Standard Time)",
        "Date2" : ISODate("2021-02-03T19:30:40.014Z"),
        "Date3" : ISODate("2021-02-03T19:30:56.454Z")
}
>
```

Key Features:

- It is used for storing JavaScript code with scope.
- it is deprecated in MongoDB 4.4.
- It Uses alternatives like aggregation pipelines or external languages for dynamic code execution.

17. Timestamp

In MongoDB, this data type is used to store a timestamp. It is useful when we modify our data to keep a record and the value of this data type is 64-bit. The value of the timestamp data type is always unique.

Example:

```
> var timestmp = new Timestamp()
> timestmp
Timestamp(0, 0)
> db.time.insertOne({Timestamp: timestmp})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("601afafc6fd54aa34c9c6dfd")
}
> db.time.find().pretty()
{
        "_id" : ObjectId("601afafc6fd54aa34c9c6dfd"),
        "Timestamp" : Timestamp(1612380924, 1)
}
>
```

In this example, the `updated_at` field stores a timestamp value.

Key Features:

- It is used for time tracking.
- It is used to store a 64-bit value

18. Decimal

This MongoDB data type store 128-bit decimal-based floating-point value. This data type was introduced in MongoDB version 3.4

Example:

```
> var decimal = NumberDecimal("1000.55")
> decimal
NumberDecimal("1000.55")
> db.decimal.insertOne({NumberDecimal: decimal})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("6299c81c2fe2f881ac89c8a6")
}
> db.decimal.find().pretty()
{
        "_id" : ObjectId("6299c81c2fe2f881ac89c8a6"),
        "NumberDecimal" : NumberDecimal("1000.55")
}
```

5.4 MONGO DB INSTALLATION

How to Install MongoDB on Windows?

Last Updated : 18 Sep, 2025

Installing MongoDB on Windows involves downloading the official MongoDB installer, running it to set up the server, and configuring it to run as a Windows service. Once installed, you can use the MongoDB shell or Compass GUI to interact with your databases. This process ensures MongoDB is ready for development or production use on a Windows system.

Requirements for Installing MongoDB on Windows

Before installing MongoDB on Windows, ensure your system meets the following requirements:

1. Supported Versions

- MongoDB 4.4 or higher (64-bit only).

2. Compatible Operating Systems

- Windows Server 2022
- Windows Server 2019
- Windows 11

2. Permissions Required

The user running MongoDB services (mongod, mongos) must have membership in the following groups:

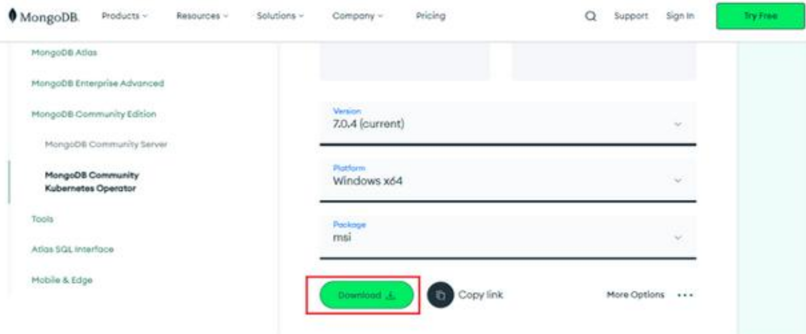
- Performance Monitor Users
- Performance Log Users

How to Install MongoDB on Windows Using MSI

To install MongoDB on Windows, first, download the MongoDB server and then install the MongoDB shell. The Steps below explain the installation process in detail and provide the required resources for the smooth download and install MongoDB.

Step 1: Download MongoDB Community Server

Go to the MongoDB Download Center to download the MongoDB Community Server.

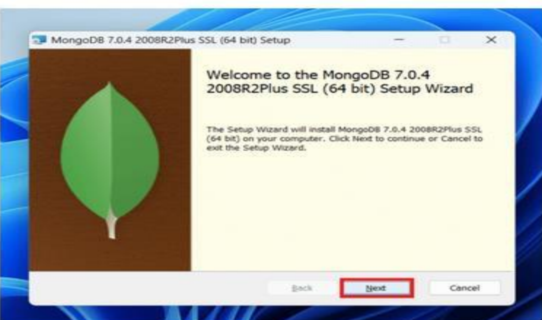


Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

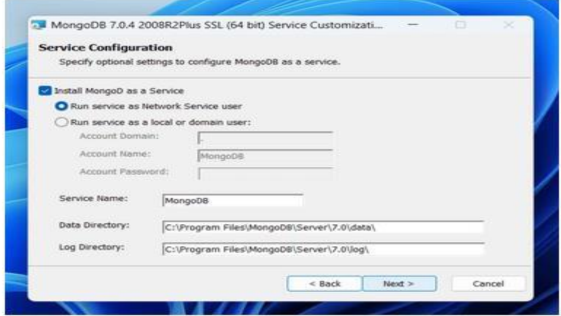
- **Version: 7.0.4**
- **OS: Windows x64**
- **Package: msi**

Step 2: Install MongoDB

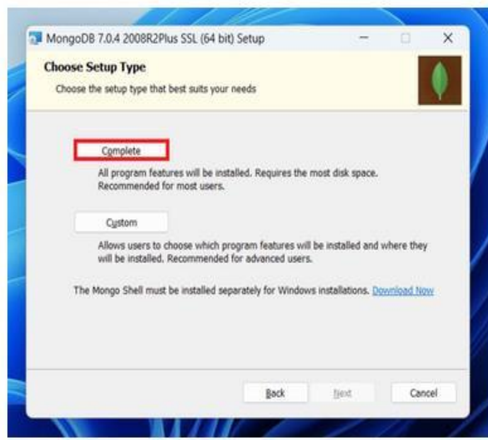
- When the download is complete open the msi file and click the next button in the startup screen:



- Now accept the End-User License Agreement and click the next button:

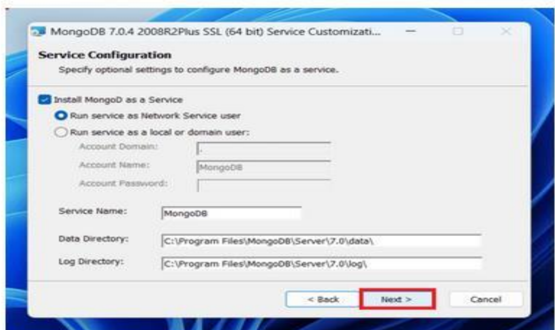


- Now select the complete option to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the Custom option:

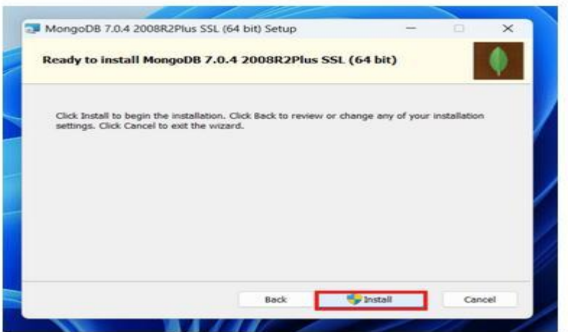


Step 3: Configure MongoDB Service

- Select "Run service as Network Service user" and copy the path of the data directory. Click Next:



- Click the Install button to start the MongoDB installation process:



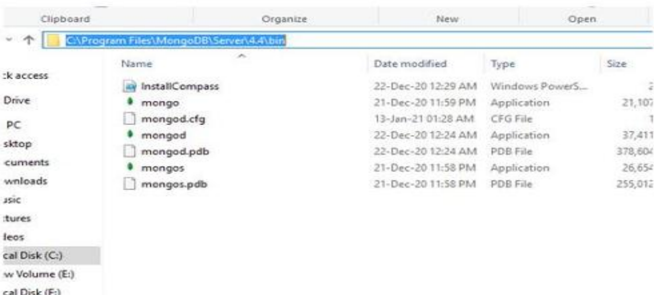
- After clicking on the install button installation of MongoDB begins:

Step 4: Complete Installation

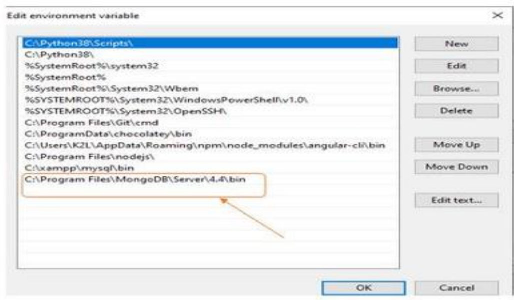
- Now click the Finish button to complete the MongoDB installation process:

Step 5: Set Environment Variables

- Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:



- Now, to create an environment variable open system properties >> Environment Variable >> System variable >> path >> Edit Environment variable
- paste the copied link to your environment system and click Ok:



Run MongoDB Server (mongod)

To run the MongoDB server (mongod) on Windows, follow these steps:

Step 1. Start MongoDB Service

- After setting the environment variable, we will run the MongoDB server, i.e. mongod.
- So, open the command prompt and run the following command:

```
mongod
```

When you run this command you will get an error i.e. C:/data/db/ not found.

Step 2. Create Required Folders

- Now, Open C drive and create a folder named "data"
- Inside the data folder create another folder named "db".

Step 3. Restart MongoDB

After creating these folders. Again open the command prompt and run the following command:

```
mongod
```

Now, this time the MongoDB server(i.e., mongod) will run successfully.

```
C:\Users\NIkhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W",  "c":"ASIO",     "id":22601,   "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23398,   "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx"
gitVersion":"913d6b62acfbb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc","
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx"
ndows 10","version":"10.0 (build 14393)"}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I",  "c":"STORAGE",  "id":22270,   "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
```

Run the MongoDB Shell (mongosh)

Starting from MongoDB version 5.0, the traditional MongoDB shell (`mongo`) has been deprecated. The recommended shell for interacting with MongoDB databases is now **mongosh**, which provides improved functionality, better syntax, and full compatibility with the latest MongoDB features.

Step 1. Connect to MongoDB Server with mongosh

- Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window
- open a new command prompt window and type:

```
mongosh
```

- You are now connected to the MongoDB shell.

Please do not close the mongod window if you close this window your server will stop working and it will not able to connect with the mongo shell.

```
C:\Users\NIkhil Chhipa>mongo
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("96cca5da-dc9f-4a40-aabb-732ee37600c0") }
MongoDB server version: 4.4.3
---
The server generated these startup warnings when booting:
        2021-01-28T20:56:52.570+05:30: Access control is not enabled for the database. Read and write access
configuration is unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Step 2. Create a Database

Now we can make a new database, collections, and documents in our shell. Use the following command within the mongosh shell to create a new database:

```
use database_name
```

The use Database_name command makes a new database in the system if it does not exist, if the database exists it uses that database:

```
use gfg
```

Step 3: Add Data to a Collection

Insert a document into a collection using:

```
db.collection_name.insertOne({field: value})
```

The db.Collection_name command makes a new collection in the gfg database and the insertOne() method inserts the document in the student collection:

```
db.student.insertOne({Akshay:500})
```

```
> use gfg
switched to db gfg
> db.student.insertOne({Akshay:500})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60083bf8b7388ed4d54157c9")
}
> db.student.find().pretty()
{ "_id" : ObjectId("60083bf8b7388ed4d54157c9"), "Akshay" : 500 }
>
```

5.5 MongoDB - Data Modelling

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collections documents may hold different types of data.

- **Data Model Design**

MongoDB provides two types of data models: Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

- **Embedded Data Model**

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below −

```
{
     _id: ,
     Emp_ID: "10025AE336"
     Personal_details:{
          First_Name: "Radhika",
          Last_Name: "Sharma",
          Date_Of_Birth: "1995-09-26"
     },
     Contact: {
          e-mail: "radhika_sharma.123@gmail.com",
          phone: "9848022338"
     },
     Address: {
          city: "Hyderabad",
          Area: "Madapur",
          State: "Telangana"
     }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
    _id: <ObjectId101>,
    Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
    _id: <ObjectId102>,
    empDocID: " ObjectId101",
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
}
```

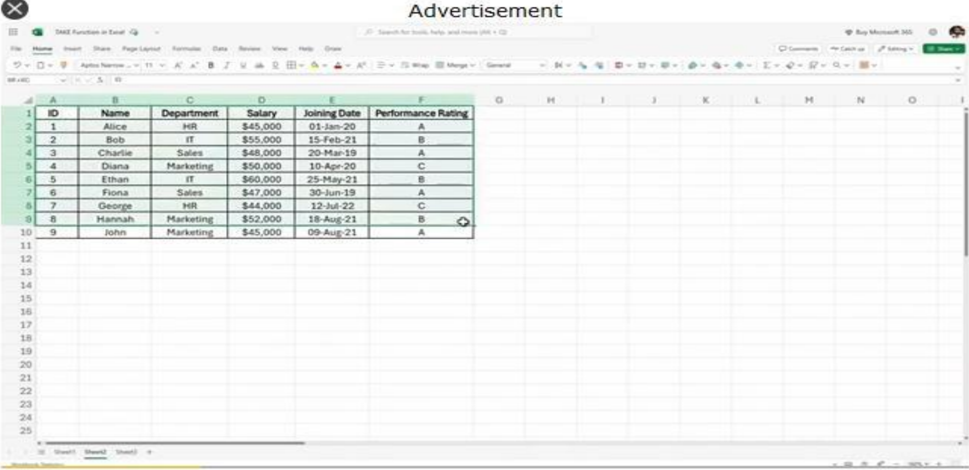
Contact:

```
{
    _id: <ObjectId103>,
    empDocID: " ObjectId101",
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
}
```

Address:

```
{
    _id: <ObjectId104>,
    empDocID: " ObjectId101",
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
}
```

Advertisement



ID	Name	Department	Salary	Joining Date	Performance Rating
1	Alice	HR	$45,000	01-Jan-20	A
2	Bob	IT	$55,000	15-Feb-21	B
3	Charlie	Sales	$48,000	20-Mar-19	A
4	Diana	Marketing	$50,000	10-Apr-20	C
5	Ethan	IT	$60,000	25-May-21	B
6	Fiona	Sales	$47,000	30-Jun-19	A
7	George	HR	$44,000	12-Jul-22	C
8	Hannah	Marketing	$52,000	18-Aug-21	B
9	John	Marketing	$45,000	09-Aug-21	A

Considerations while designing Schema in MongoDB

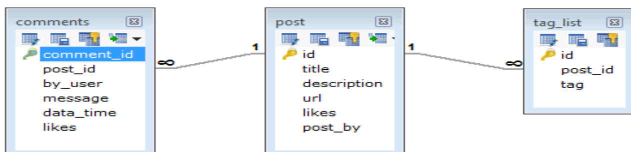
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure –

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

5.6 MongoDB - Create Database

The use Command

MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows –

```
use DATABASE_NAME
```

Example

If you want to use a database with name **<mydb>**, then **use DATABASE** statement would be as follows –

```
>use mydb
switched to db mydb
```

To check your currently selected database, use the command **db**

```
>db
mydb
```

If you want to check your databases list, use the command **show dbs**.

```
>show dbs
local     0.78125GB
test      0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

5.7MongoDB - Drop Database

The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows –

```
db.dropDatabase()
```

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

First, check the list of available databases by using the command, **show dbs**.

```
>show dbs
local      0.78125GB
mydb       0.23012GB
test       0.23012GB
>
```

If you want to delete new database **<mydb>**, then **dropDatabase()** command would be as follows –

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local      0.78125GB
test       0.23012GB
>
```

5.8 MongoDB - Insert Document

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

Syntax

The basic syntax of **insert()** command is as follows −

```
>db.COLLECTION_NAME.insert(document)
```

Example

```
> db.users.insert({
... _id : ObjectId("507f191e810c19729de860ea"),
... title: "MongoDB Overview",
... description: "MongoDB is no sql database",
... by: "tutorials point",
... url: "http://www.tutorialspoint.com",
... tags: ['mongodb', 'database', 'NoSQL'],
... likes: 100
... })
WriteResult({ "nInserted" : 1 })
>
```

Here **mycol** is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document.

_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows –

```
_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)
```

You can also pass an array of documents into the insert() method as shown below:.

```
> db.createCollection("post")
> db.post.insert([
    {
        title: "MongoDB Overview",
        description: "MongoDB is no SQL database",
        by: "tutorials point",
        url: "http://www.tutorialspoint.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 100
    },
    {
    title: "NoSQL Database",
    description: "NoSQL database doesn't have tables",
    by: "tutorials point",
    url: "http://www.tutorialspoint.com",
    tags: ["mongodb", "database", "NoSQL"],
    likes: 20,
    comments: [
```

To insert the document you can use **db.post.save(document)** also. If you don't specify **_id** in the document then **save()** method will work same as **insert()** method. If you specify _id then it will replace whole data of document containing _id as specified in save() method.

Advertisement

The insertOne() method

If you need to insert only one document into a collection you can use this method.

Syntax

The basic syntax of insert() command is as follows −

```
>db.COLLECTION_NAME.insertOne(document)
```

Example

Following example creates a new collection named empDetails and inserts a document using the insertOne() method.

```
> db.createCollection("empDetails")
{ "ok" : 1 }
```

```
> db.empDetails.insertOne(
	{
		First_Name: "Radhika",
		Last_Name: "Sharma",
		Date_Of_Birth: "1995-09-26",
		e_mail: "radhika_sharma.123@gmail.com",
		phone: "9848022338"
	})
{
	"acknowledged" : true,
	"insertedId" : ObjectId("5dd62b4070fb13eec3963bea")
}
>
```

The insertMany() method

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

Example

Following example inserts three different documents into the empDetails collection using the insertMany() method.

```
> db.empDetails.insertMany(
	[
		{
			First_Name: "Radhika",
			Last_Name: "Sharma",
			Date_Of_Birth: "1995-09-26",
			e_mail: "radhika_sharma.123@gmail.com",
			phone: "9000012345"
		},
		{
			First_Name: "Rachel",
			Last_Name: "Christopher",
			Date_Of_Birth: "1990-02-16",
			e_mail: "Rachel_Christopher.123@gmail.com",
			phone: "9000054321"
		},
		{
			First_Name: "Fathima",
```

5.9 MongoDB Find

Find Data

There are 2 methods to find and select data from a MongoDB collection, `find()` and `findOne()`.

`find()`

To select data from a collection in MongoDB, we can use the `find()` method.

This method accepts a query object. If left empty, all documents will be returned.

Example

```
db.posts.find()
```

Try it Yourself »

`findOne()`

To select only one document, we can use the `findOne()` method.

This method accepts a query object. If left empty, it will return the first document it finds.

Querying Data

To query, or filter, data we can include a query in our `find()` or `findOne()` methods.

Example

```
db.posts.find( {category: "News"} )
```

Try it Yourself »

Projection

Both find methods accept a second parameter called `projection`.

This parameter is an `object` that describes which fields to include in the results.

Note: This parameter is optional. If omitted, all fields will be included in the results.

Example

This example will only display the `title` and `date` fields in the results.

```
db.posts.find({}, {title: 1, date: 1})
```

Notice that the _id field is also included. This field is always included unless specifically excluded.

We use a 1 to include a field and 0 to exclude a field.

Example

This time, let's exclude the _id field.

```
db.posts.find({}, {_id: 0, title: 1, date: 1})
```

Try it Yourself »

Note: You cannot use both 0 and 1 in the same object. The only exception is the _id field. You should either specify the fields you would like to include or the fields you would like to exclude.

Let's exclude the date category field. All other fields will be included in the results.

Example

```
db.posts.find({}, {category: 0})
```

Try it Yourself »

We will get an error if we try to specify both 0 and 1 in the same object.

Example

```
db.posts.find({}, {title: 1, date: 0})
```

5.10 MongoDB - Update Document

MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of **update()** method is as follows –

```
>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
```

Example

Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

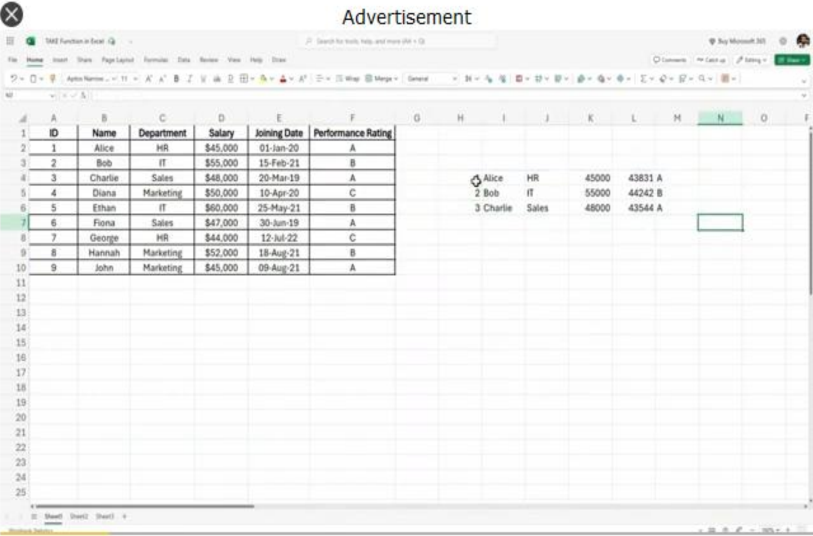
Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

```
>db.mycol.update({'title':'MongoDB Overview'},
   {$set:{'title':'New MongoDB Tutorial'}},{multi:true})
```

Advertisement



MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB **save()** method is shown below −

```
>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})
```

Example

Following example will replace the document with the _id '5983548781331adf45ec5'.

```
>db.mycol.save(
   {
      "_id" : ObjectId("507f191e810c19729de860ea"),
        "title":"Tutorials Point New Topic",
      "by":"Tutorials Point"
   }
)
WriteResult({
     "nMatched" : 0,
     "nUpserted" : 1,
     "nModified" : 0,
     "_id" : ObjectId("507f191e810c19729de860ea")
})
>db.mycol.find()
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point New Topic",
   "by":"Tutorials Point"}
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"NoSQL Overview"}
{ "_id" : ObjectId("507f191e810c19729de860e6"), "title":"Tutorials Point Overview"}
```

MongoDB findOneAndUpdate() method

The **findOneAndUpdate()** method updates the values in the existing document.

Syntax

The basic syntax of **findOneAndUpdate()** method is as follows −

```
>db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)
```

Example

Assume we have created a collection named empDetails and inserted three documents in it as shown below −

```
> db.empDetails.insertMany(
    [
        {
            First_Name: "Radhika",
            Last_Name: "Sharma",
            Age: "26",
            e_mail: "radhika_sharma.123@gmail.com",
            phone: "9000012345"
        },
        {
            First_Name: "Rachel",
            Last_Name: "Christopher",
            Age: "27",
            e_mail: "Rachel_Christopher.123@gmail.com",
            phone: "9000054321"
        },
        {
            First_Name: "Fathima",
```

Following example updates the age and email values of the document with name 'Radhika'.

```
> db.empDetails.findOneAndUpdate(
    {First_Name: 'Radhika'},
    { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}}
)
{
    "_id" : ObjectId("5dd6636870fb13eec3963bf5"),
    "First_Name" : "Radhika",
    "Last_Name" : "Sharma",
    "Age" : "30",
    "e_mail" : "radhika_newemail@gmail.com",
    "phone" : "9000012345"
}
```

MongoDB updateOne() method

This methods updates a single document which matches the given filter.

Syntax

The basic syntax of updateOne() method is as follows −

```
>db.COLLECTION_NAME.updateOne(<filter>, <update>)
```

Example

```
> db.empDetails.updateOne(
    {First_Name: 'Radhika'},
    { $set: { Age: '30',e_mail: 'radhika_newemail@gmail.com'}}
)
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
>
```

MongoDB updateMany() method

The updateMany() method updates all the documents that matches the given filter.

Syntax

The basic syntax of updateMany() method is as follows –

```
>db.COLLECTION_NAME.update(<filter>, <update>)
```

Example

```
> db.empDetails.updateMany(
    {Age:{ $gt: "25" }},
    { $set: { Age: '00'}}
)
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

You can see the updated values if you retrieve the contents of the document using the find method as shown below –

```
> db.empDetails.find()
{ "_id" : ObjectId("5dd6636870fb13eec3963bf5"), "First_Name" : "Radhika", "Last_Name" : "Sharma", "Age" :
"00", "e_mail" : "radhika_newemail@gmail.com", "phone" : "9000012345" }
{ "_id" : ObjectId("5dd6636870fb13eec3963bf6"), "First_Name" : "Rachel", "Last_Name" : "Christopher",
"Age" : "00", "e_mail" : "Rachel_Christopher.123@gmail.com", "phone" : "9000054321" }
{ "_id" : ObjectId("5dd6636870fb13eec3963bf7"), "First_Name" : "Fathima", "Last_Name" : "Sheik", "Age" :
"24", "e_mail" : "Fathima_Sheik.123@gmail.com", "phone" : "9000054321" }
>
```

5.11 MongoDB - Delete Document

The remove() Method

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria** – (Optional) deletion criteria according to documents will be removed.
- **justOne** – (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of **remove()** method is as follows –

```
>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
```

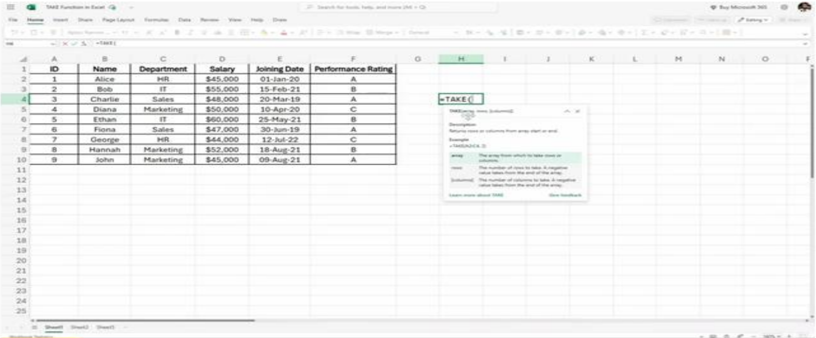
Example

Consider the mycol collection has the following data.

```
{_id : ObjectId("507f191e810c19729de860e1"), title: "MongoDB Overview"},
{_id : ObjectId("507f191e810c19729de860e2"), title: "NoSQL Overview"},
{_id : ObjectId("507f191e810c19729de860e3"), title: "Tutorials Point Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'.

```
>db.mycol.remove({'title':'MongoDB Overview'})
WriteResult({"nRemoved" : 1})
> db.mycol.find()
{"_id" : ObjectId("507f191e810c19729de860e2"), "title" : "NoSQL Overview" }
{"_id" : ObjectId("507f191e810c19729de860e3"), "title" : "Tutorials Point Overview" }
```



ID	Name	Department	Salary	Joining Date	Performance Rating
1	Alice	HR	$45,000	01-Jan-20	A
2	Bob	IT	$55,000	15-Feb-21	B
3	Charlie	Sales	$48,000	20-Mar-19	A
4	Diana	Marketing	$50,000	10-Apr-20	C
5	Ethan	IT	$60,000	25-May-21	B
6	Fiona	Sales	$47,000	30-Jun-19	A
7	George	HR	$44,000	12-Jul-22	C
8	Hannah	Marketing	$52,000	18-Aug-21	B
9	John	Marketing	$45,000	09-Aug-21	A

Remove Only One

If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

```
>db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)
```

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

```
> db.mycol.remove({})
WriteResult({ "nRemoved" : 2 })
> db.mycol.find()
>
```

5.12 MongoDB - Query Document

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows −

```
>db.COLLECTION_NAME.find()
```

find() method will display all the documents in a non-structured way.

Example

Assume we have created a collection named mycol as −

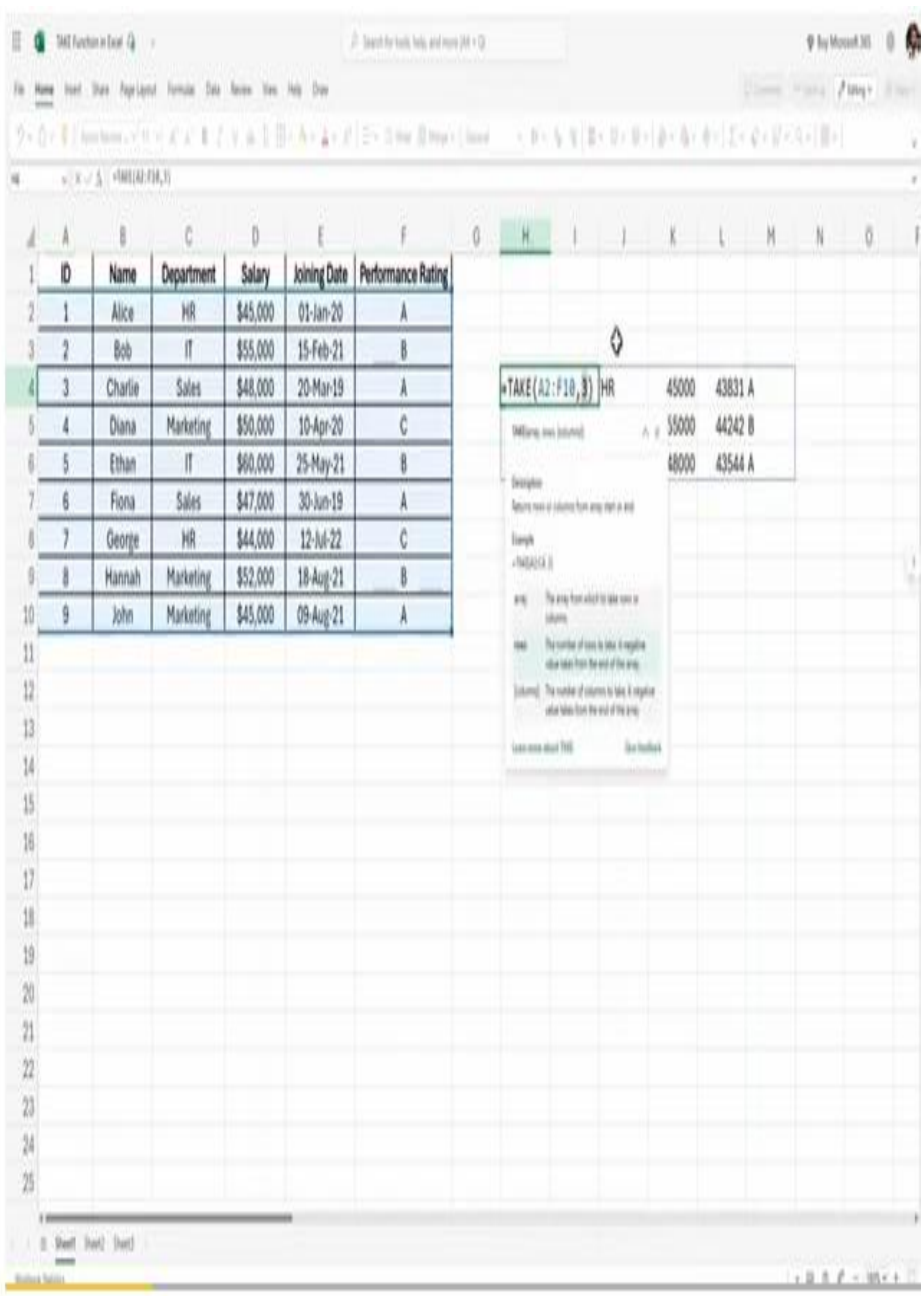
```
> use sampleDB
switched to db sampleDB
> db.createCollection("mycol")
{ "ok" : 1 }
>
```

And inserted 3 documents in it using the insert() method as shown below –

```
> db.mycol.insert([
    {
        title: "MongoDB Overview",
        description: "MongoDB is no SQL database",
        by: "tutorials point",
        url: "http://www.tutorialspoint.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 100
    },
    {
        title: "NoSQL Database",
        description: "NoSQL database doesn't have tables",
        by: "tutorials point",
        url: "http://www.tutorialspoint.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 20,
        comments: [
            {
```

Following method retrieves all the documents in the collection –

```
> db.mycol.find()
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534c"), "title" : "MongoDB Overview", "description" : "MongoDB is no SQL database", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 100 }
{ "_id" : ObjectId("5dd4e2cc0821d3b44607534d"), "title" : "NoSQL Database", "description" : "NoSQL database doesn't have tables", "by" : "tutorials point", "url" : "http://www.tutorialspoint.com", "tags" : [ "mongodb", "database", "NoSQL" ], "likes" : 20, "comments" : [ { "user" : "user1", "message" : "My first comment", "dateCreated" : ISODate("2013-12-09T21:05:00Z"), "like" : 0 } ] }
>
```



ID
Name
Department
Salary
Joining Date
Performance Rating
1 Alice HR $45,000 01-Jan-20 A
2 Bob IT $55,000 15-Feb-21 B
3 Charlie Sales $48,000 20-Mar-19 A
4 Diana Marketing $50,000 10-Apr-20 C
5 Ethan IT $60,000 25-May-21 B
6 Fiona Sales $47,000 30-Jun-19 A
7 George HR $44,000 12-Jul-22 C
8 Hannah Marketing $52,000 18-Aug-21 B
9 John Marketing $45,000 09-Aug-21 A
=TAKE(A2:F10,3)
HR
45000 43831 A
55000 44242 B
48000 43544 A

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

Example

Following example retrieves all the documents from the collection named mycol and arranges them in an easy-to-read format.

```
> db.mycol.find().pretty()
{
	"_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
	"title" : "MongoDB Overview",
	"description" : "MongoDB is no SQL database",
	"by" : "tutorials point",
	"url" : "http://www.tutorialspoint.com",
	"tags" : [
		"mongodb",
		"database",
		"NoSQL"
	],
	"likes" : 100
}
{
	"_id" : ObjectId("5dd4e2cc0821d3b44607534d"),
	"title" : "NoSQL Database",
	"description" : "NoSQL database doesn't have tables",
```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

Example

Following example retrieves the document with title MongoDB Overview.

```
> db.mycol.findOne({title: "MongoDB Overview"})
{
	"_id" : ObjectId("5dd6542170fb13eec3963bf0"),
	"title" : "MongoDB Overview",
	"description" : "MongoDB is no SQL database",
	"by" : "tutorials point",
	"url" : "http://www.tutorialspoint.com",
	"tags" : [
		"mongodb",
		"database",
		"NoSQL"
	],
	"likes" : 100
}
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>, <valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesnt exist at all

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND –

```
>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })
```

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
> db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
    "_id" : ObjectId("5dd4e2cc0821d3b44607534c"),
    "title" : "MongoDB Overview",
    "description" : "MongoDB is no SQL database",
    "by" : "tutorials point",
    "url" : "http://www.tutorialspoint.com",
    "tags" : [
        "mongodb",
        "database",
        "NoSQL"
    ],
    "likes" : 100
}
>
```

For the above given example, equivalent where clause will be **' where by = 'tutorials point' AND title = 'MongoDB Overview' '**. You can pass any number of key, value pairs in find clause.

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** –

```
>db.mycol.find(
   {
      $or: [
         {key1: value1}, {key2:value2}
      ]
   }
).pretty()
```

Example

Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Using AND and OR Together

Example

The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is **'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'**

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT**

```
>db.COLLECTION_NAME.find(
    {
        $not: [
            {key1: value1}, {key2:value2}
        ]
    }
)
```

Example

Assume we have inserted 3 documents in the collection **empDetails** as shown below −

```
            phone: "9000012345"
        },
        {
            First_Name: "Rachel",
            Last_Name: "Christopher",
            Age: "27",
            e_mail: "Rachel_Christopher.123@gmail.com",
            phone: "9000054321"
        },
        {
            First_Name: "Fathima",
            Last_Name: "Sheik",
            Age: "24",
            e_mail: "Fathima_Sheik.123@gmail.com",
            phone: "9000054321"
        }
```

```
        {
            First_Name: "Fathima",
            Last_Name: "Sheik",
            Age: "24",
            e_mail: "Fathima_Sheik.123@gmail.com",
            phone: "9000054321"
        }
    ]
)
```

Following example will retrieve the document(s) whose first name is not "Radhika" and last name is not "Christopher"

```
> db.empDetails.find(
    {
        $nor:[
            40
            {"First_Name": "Radhika"},
            {"Last_Name": "Christopher"}
        ]
    }
).pretty()
{
    "_id" : ObjectId("5dd631f270fb13eec3963bef"),
    "First_Name" : "Fathima",
    "Last_Name" : "Sheik",
    "Age" : "24",
    "e_mail" : "Fathima_Sheik.123@gmail.com",
    "phone" : "9000054321"
}
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT**

```
>db.COLLECTION_NAME.find(
    {
        $NOT: [
            {key1: value1}, {key2:value2}
        ]
    }
).pretty()
```

Example

Following example will retrieve the document(s) whose age is not greater than 25

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
{
    "_id" : ObjectId("5dd6636870fb13eec3963bf7"),
    "First_Name" : "Fathima",
    "Last_Name" : "Sheik",
    "Age" : "24",
    "e_mail" : "Fathima_Sheik.123@gmail.com",
    "phone" : "9000054321"
}
```