

Unit 1: Programming in Java

Presented by: Aakash Raj Shakya





Introduction to Java

1. Java is a popular programming language, created in 1995 by James Gosling from Sun Microsystems (Sun).
2. The target of Java is to write a program once and then run this program on multiple operating systems.
3. The first publicly available version of Java (Java 1.0) was released in 1995.
4. It was owned by Oracle in 2010, and more than 3 billion devices run Java.
5. Latest version of Java is **Java 17 or JDK 17**.

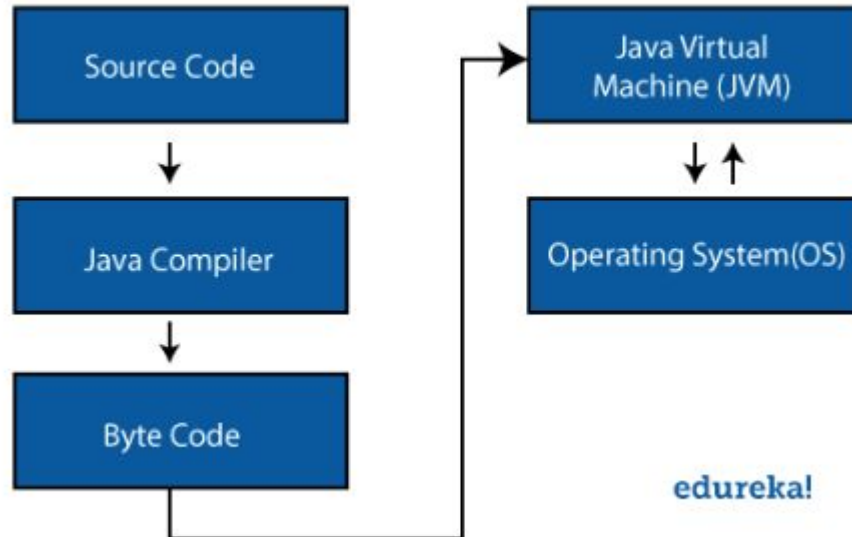




Java Architecture

Java Architecture combines the process of compilation and interpretation. It explains the various processes involved whilst formulating a Java program.

1. In Java, there is a process of compilation and interpretation.
2. The code written in Java, is converted into byte codes which is done by the Java Compiler.
3. The byte codes, then are converted into machine code by the JVM.
4. The Machine code is executed directly by the machine.



edureka!



Components of Java Architecture

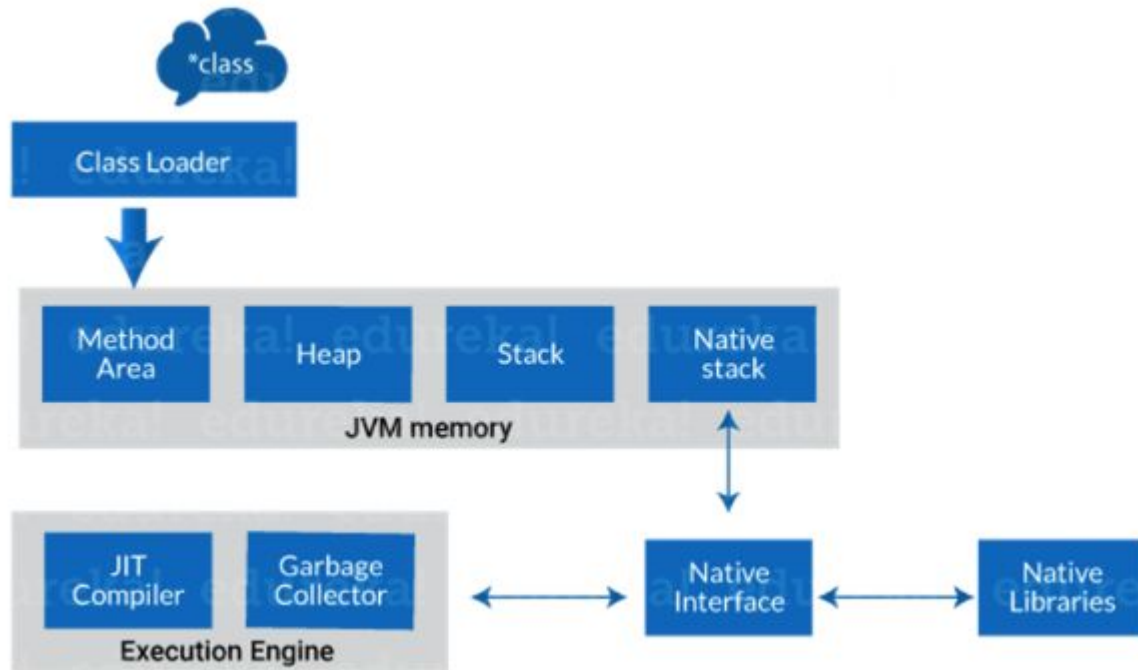
There are three main components of Java language:

1. Java Virtual Machine (**JVM**)
2. Java Runtime Environment (**JRE**)
3. Java Development Kit (**JDK**)



Java Virtual Machine (JVM)

1. Java applications are called WORA (Write Once Run Anywhere) because of their ability to run a code on any platform. This is done only because of JVM.
2. The JVM is a Java platform component that provides an environment for executing Java programs.
3. JVM interprets the bytecode into machine code which is executed in the machine in which the Java program runs.
4. In a nutshell JVM performs the following functions:
 - a. Loads the code
 - b. Verifies the code
 - c. Executes the code
 - d. Provides runtime environment





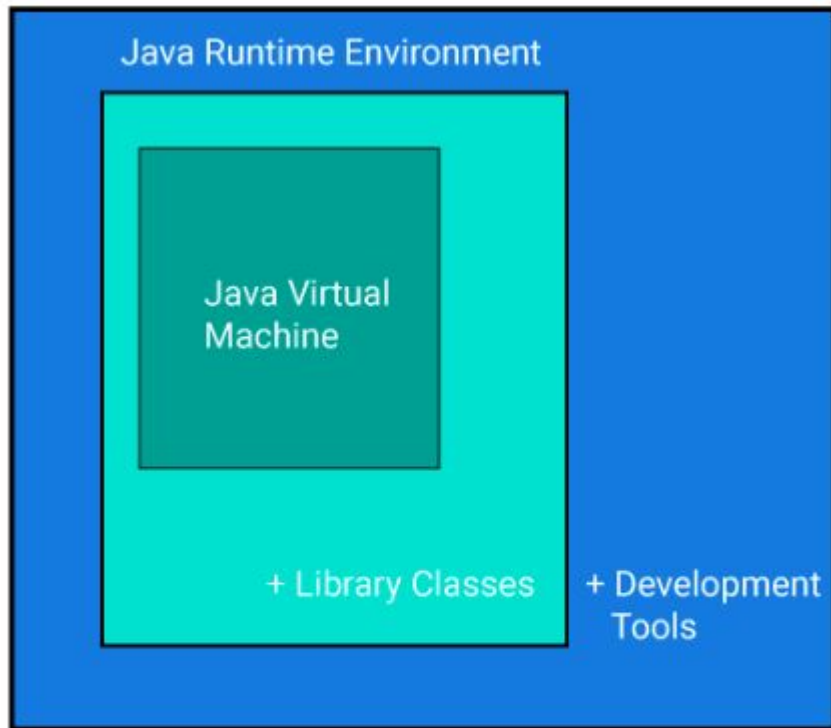
Java Runtime Environment (JRE)

1. The JRE software builds a runtime environment in which Java programs can be executed.
2. The JRE is the on-disk system that takes your Java code, combines it with the needed libraries, and starts the JVM to execute it.
3. The JRE contains libraries and software needed by your Java programs to run. JRE is a part of JDK (which we will study later) but can be downloaded separately.



Java Development Kit (JDK)

1. The Java Development Kit (JDK) is a software development environment used to develop Java applications and applets.
2. It contains JRE and several development tools, an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) accompanied with another tool.



JDK = JRE + Development Tool

JRE = JVM + Library Classes

edureka!



<https://www.edureka.co/blog/java-architecture/>



Advantages of Java

1. Simple and familiar
2. Object Oriented
3. Platform Independent
4. Architecture Neutral
 - a. Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
 - b. In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.
5. Distributed
6. Multi-threaded
7. Robust
8. Dynamic and Extensible: Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).



9. Secure: Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
- a. No explicit pointer
 - b. Java Programs run inside a virtual machine sandbox
 - c. Classloader: Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
 - d. Bytecode Verifier: It checks the code fragments for illegal code that can violate access right to objects.
 - e. Security Manager: It determines what resources a class can access such as reading and writing to the local disk.



Class

- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- The class forms the basis for object-oriented programming in Java.
- Any concept you wish to implement in a Java program must be encapsulated within a class.
- The most important thing to understand about a class is that it defines a new data type.
- Once defined, this new type can be used to create objects of that type.
- Thus, a class is a template for an object, and an object is an instance of a class.
- Because an object is an instance of a class



Syntax of Java Class

```
public class ClassName {  
  
    //Class variables  
  
    //Class methods.  
  
}
```




Example:

```
Public class Student {
```

```
    private String name;
```

```
    private int age;
```

```
    Public String getStudentInfo(){
```

```
        return "Student name is " + name + " and age is " + age;
```

```
    }
```

```
}
```



Creating Objects

1. When you create a class, you are creating a new data type.
2. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process.
3. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
4. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.
5. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
6. This reference is, more or less, the address in memory of the object allocated by new.
7. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.



Different ways of creating objects:

```
Student student = new Student();
```

Or

```
Student student;
```

```
student = new Student();
```



Constructors

1. Java allows objects to initialize themselves when they are created.
2. This automatic initialization is performed through the use of a constructor.
3. A constructor initializes an object immediately upon creation.
4. It has the same name as the class in which it resides and is syntactically similar to a method.
5. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.



Types of constructor

1. Default Constructor
 - a. A constructor is called "Default Constructor" when it doesn't have any parameter.
2. No args constructor
 - a. Constructor with no arguments is known as no-arg constructor.
 - b. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.
 - c. Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write `public Demo() { }` in your class Demo it cannot be called default constructor since you have written the code of it.
3. Parameterized Constructor
 - a. Constructor with arguments(or you can say parameters) is known as Parameterized constructor.

No args Constructor:

```
public class Student {  
    private String name;  
    private int age;  
  
    public Student() {  
    }  
}
```

Parameterized Constructor:

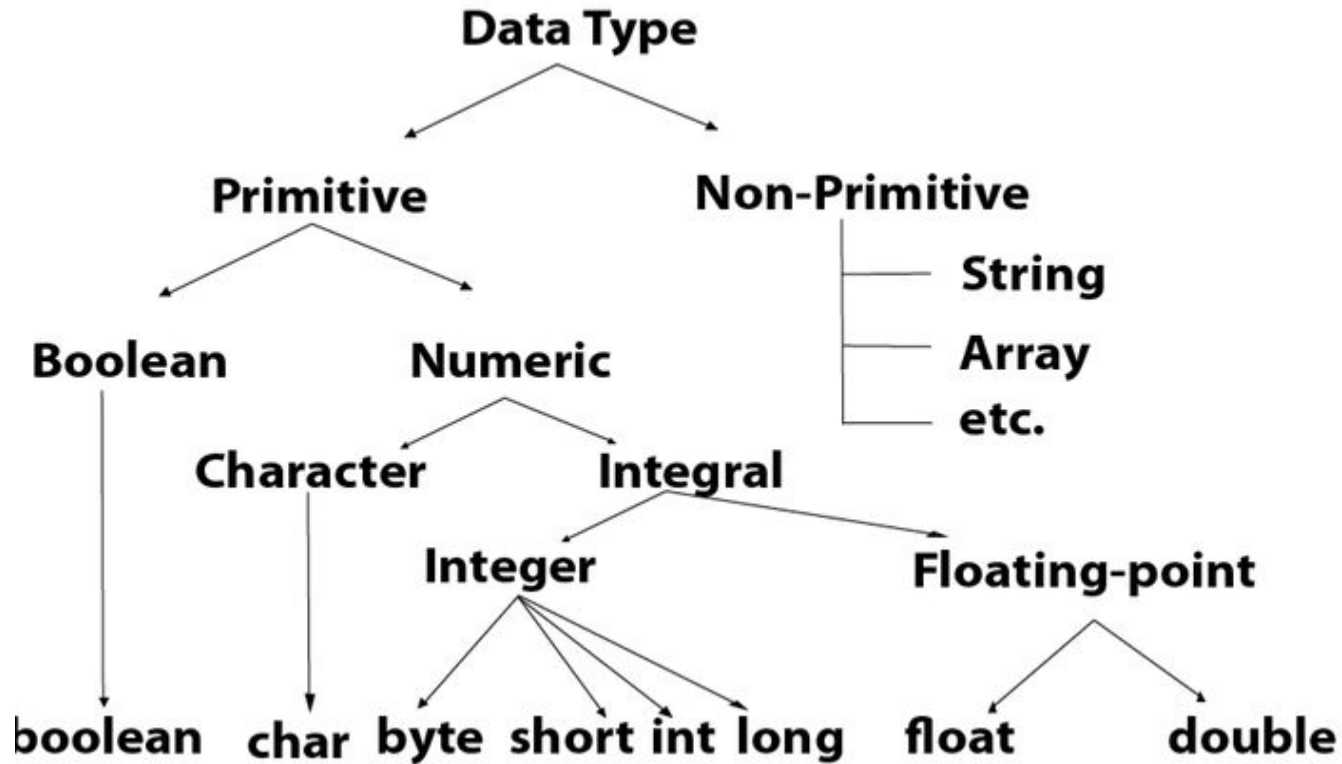
```
public class Student {  
    private String name;  
    private int age;  
  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```



Data types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. Primitive data types:
The primitive data types include **boolean, char, byte, short, int, long, float and double.**
2. Non-primitive data types:
The non-primitive data types include **Classes, Interfaces, and Arrays.**





Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte



final keyword in Java

1. final keyword is used in different contexts.
2. First of all, final is a non-access modifier applicable only to a variable, a method or a class.
3. Following are different contexts where final is used.
 - a. **final variables**
 - i. To create constant variables.
 - ii. We cannot change the value of a final variable once it is initialized.
 - b. **final methods:**
 - i. Prevent method overriding
 - ii. Which means even though a subclass can call the final method of parent class without any issues but it cannot override it.
 - c. **final classes:**
 - i. Prevent inheritance.
 - ii. We cannot extend a final class.



Packages in Java

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

1. Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.iims.Employee and college.achs.Employee
2. Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
3. Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
4. Packages can be considered as data encapsulation (or data-hiding).



Abstraction in Java

1. Data abstraction is the process of hiding certain details and showing only essential information to the user.
2. Abstraction can be achieved with either abstract classes or interfaces.
3. The abstract keyword is a non-access modifier, used for classes and methods.
4. **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
5. **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

***Note:** An abstract class can have both abstract and regular methods.



```
public abstract class Shape {  
    public abstract void area();  
  
    public void sleep() {  
        System.out.println("This is method inside abstract class.");  
    }  
}
```

What we cannot do:

```
Rectangle rectangle = new Shape();    //This cannot be done in abstract class
```



Interface in Java

1. An interface is a reference type in Java.
2. It is similar to class. It is a collection of abstract methods.
3. A class implements an interface, thereby inheriting the abstract methods of the interface.
4. Writing an interface is similar to writing a class.
 - a. But a class describes the attributes and behaviors of an object.
 - b. And an interface contains behaviors that a class implements.
5. An interface does not contain any constructors.
6. All of the methods in an interface are abstract.
7. An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
8. An interface is not extended by a class; it is implemented by a class.
9. An interface can extend multiple interfaces.



```
public interface Animal {
```

```
    public void eat();
```

```
}
```

```
//Implementing interface
```

```
public class Dog implements Animal {
```

```
    @Override
```

```
    public void eat(){
```

```
        System.out.println("Dogs are omnivores in nature.");
```

```
    }
```

```
}
```



```
public class Tiger implements Animal {
```

```
    @Override
```

```
    public void eat() {
```

```
        System.out.println("Tigers are carnivore in nature.");
```

```
    }
```

```
}
```




Access Modifiers in Java

1. The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.
2. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.
3. There are four types of Java access modifiers:
 - a. **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
 - b. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
 - c. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
 - d. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.



Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y



Inheritance in Java

1. Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
2. It is an important part of OOPs (Object Oriented programming system).
3. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
4. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
5. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.



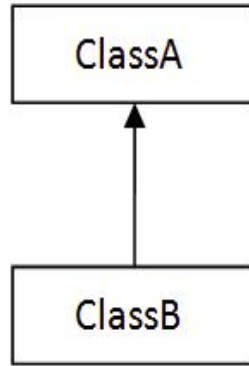
Why use inheritance in java?

1. For Method Overriding (so runtime polymorphism can be achieved).
2. For Code Reusability.

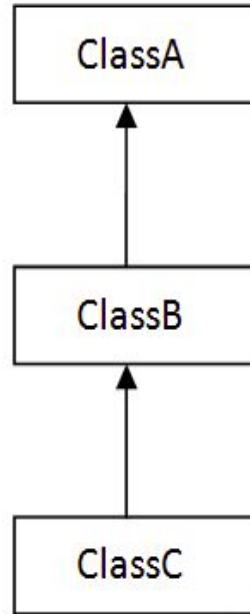
Terms used in Inheritance:

1. **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
2. **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
3. **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
4. **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

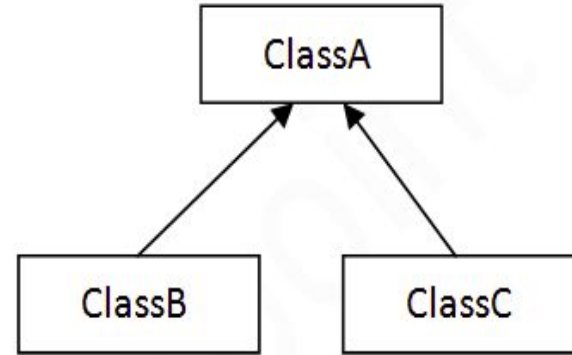
Types of Inheritance in Java



1) Single

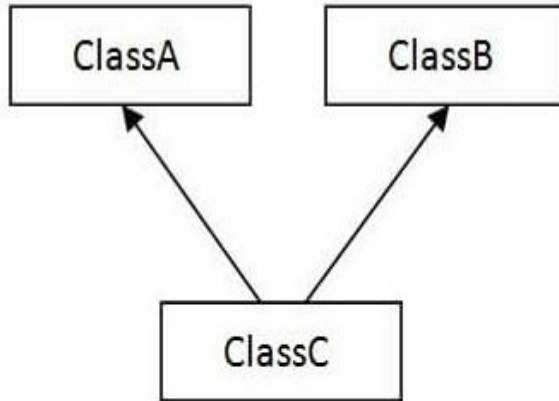


2) Multilevel

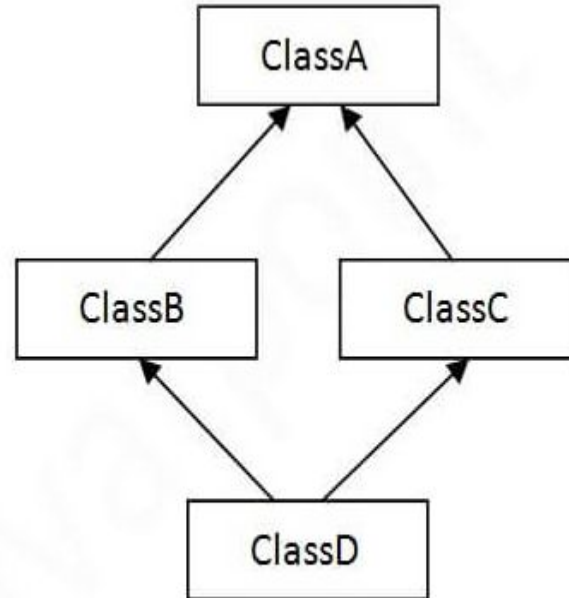


3) Hierarchical

NOT SUPPORTED IN JAVA



4) Multiple



5) Hybrid



Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

Examples:

```
String[] cars =;
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
int[] myNum = {10, 20, 30, 40};
```



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:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
System.out.println(cars[0]);
```

```
// Outputs Volvo
```


Loop Through an Array



You can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run. The following example outputs all elements in the cars array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
for (int i = 0; i < cars.length; i++) {
```

```
    System.out.println(cars[i]);
```

```
}
```

Loop Through an Array with For-Each

There is also a "for-each" loop, which is used exclusively to loop through elements in arrays:

```
for (type variable : arrayname) {
```

```
    ...
```

```
}
```



```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
for (String i : cars) {
```

```
    System.out.println(i);
```

```
}
```



Multi Dimensional Array

Multidimensional Arrays

A multidimensional array is an array containing one or more arrays.

To create a two-dimensional array, add each array within its own set of curly braces:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

```
int x = myNumbers[1][2];
```

```
System.out.println(x);
```



```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

```
for (int i = 0; i < myNumbers.length; ++i) {
```

```
    for(int j = 0; j < myNumbers[i].length; ++j) {
```

```
        System.out.println(myNumbers[i][j]);
```

```
    }
```

```
}
```



Exception Handling

An exception (or exceptional event) is a problem that arises during the execution of a program.

When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

1. A user has entered an invalid data.
2. A file that needs to be opened cannot be found.
3. A network connection has been lost in the middle of communications or the JVM has run out of memory.

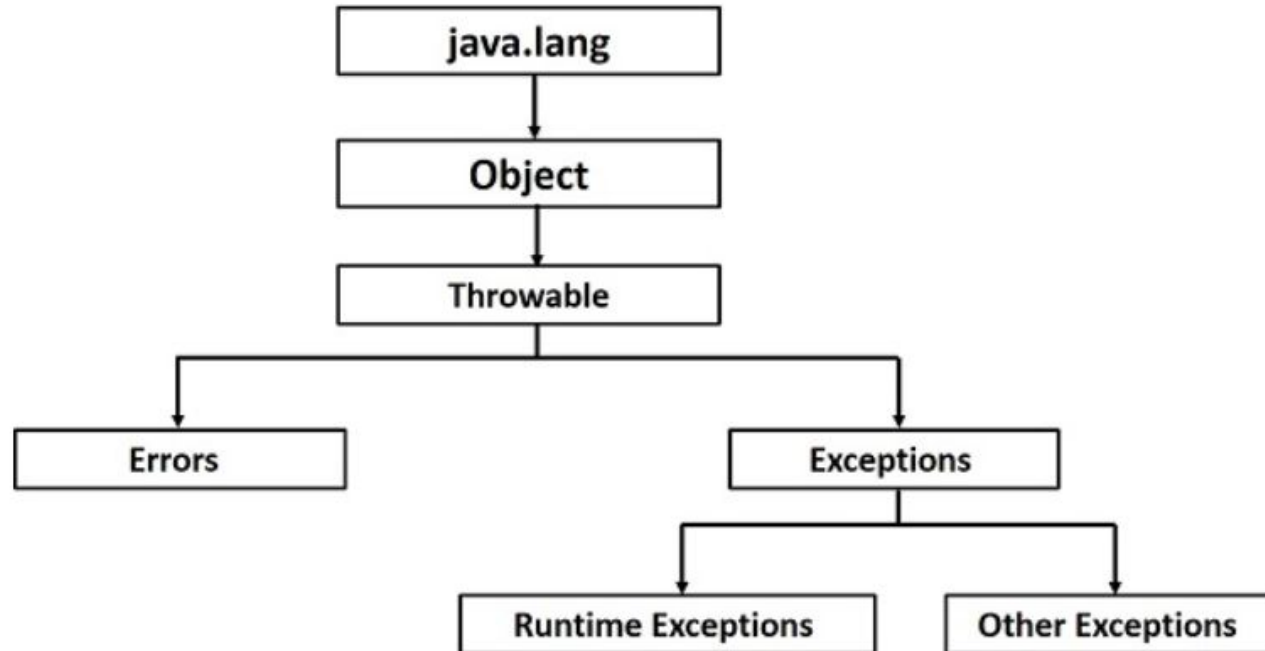


Exception Categories

1. **Checked exceptions:** A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.
2. **Unchecked exceptions:** An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
3. **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.



Exception Hierarchy





Handling exception

Major 5 keywords are used during the exception handling:

1. try
2. catch
3. throw
4. throws
5. finally



Using try/catch

A method catches an exception using a combination of the try and catch keywords.

A try/catch block is placed around the code that might generate an exception.

Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

Syntax

```
try {  
    // Protected code  
} catch (ExceptionName e1) {  
    // Catch block  
}
```



Using multiple catch

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}
```



Catching multiple type of exception

Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code.

Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1|ExceptionType2 ex) {  
    // Catch block  
}
```



The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.

```
public void fetchCarDetailsByIndex(int index) throws CustomException {  
    if(index > (cars.length - 1)){  
        throw new CustomException("The maximum size limit is " + (cars.length - 1));  
    }  
    System.out.println("The car you searched for is " + cars[index]);  
}
```



finally block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:



Syntax

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}finally {  
    // The finally block always executes.  
}
```



Multi-Threading

Unlike most other computer languages, Java provides built-in support for multithreaded programming.

A multithreaded program contains two or more parts that can run concurrently.

Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.



By definition, multitasking is when multiple processes share common processing resources such as a CPU.

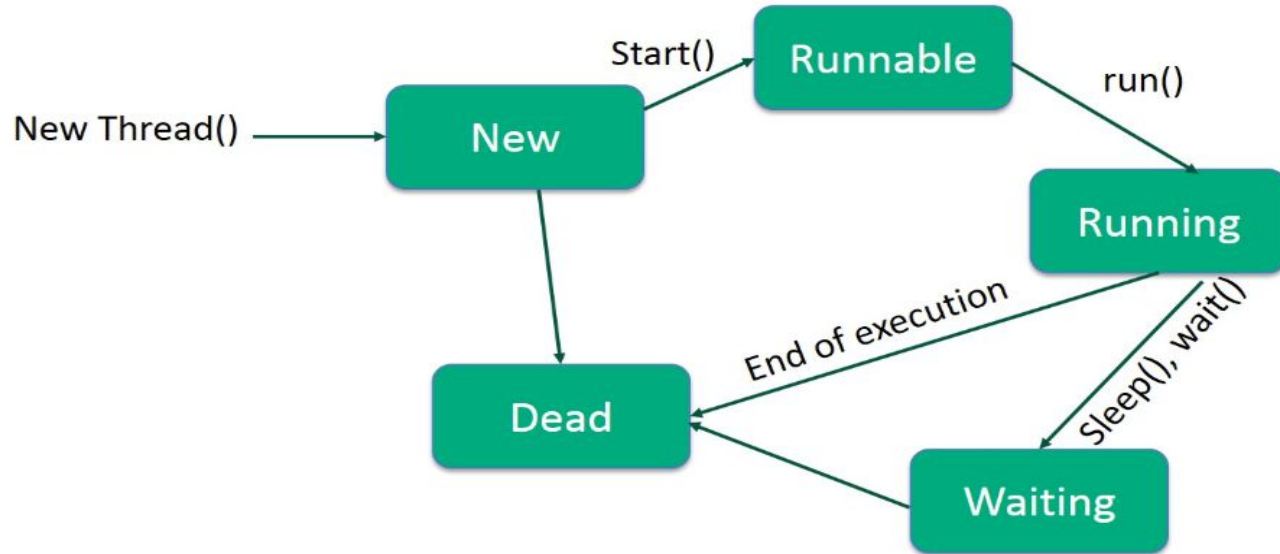
Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.

Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.



Life Cycle of Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.





Following are the stages of the life cycle –

1. **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
2. **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
3. **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
4. **Timed Waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
5. **Terminated (Dead):** A runnable thread enters the terminated state when it completes its task or otherwise terminates.



The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.

To create a new thread, program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:



Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.



The Main Thread

When a Java program starts up, one thread begins running immediately.

This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

1. It is the thread from which other “child” threads will be spawned.
2. Often it must be the last thread to finish execution because it performs various shutdown actions

Although the main thread is created automatically when program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a public static member of Thread.



Creating a thread

In the most general sense, you create a thread by instantiating an object of type `Thread`.

Java defines two ways in which this can be accomplished:

1. You can implement the `Runnable` interface.
2. You can extend the `Thread` class, itself.

The following two sections look at each method, in turn.



Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called `run()`, which is declared like this:


```
public void run( )
```

Inside `run()`, you will define the code that constitutes the new thread. It is important to understand that `run()` can call other methods, use other classes, and declare variables, just like the main thread can.



After you create a class that implements `Runnable`, you will instantiate an object of type `Thread` from within that class.

After the new thread is created, it will not start running until you call its `start()` method, which is declared within `Thread`. In essence, `start()` executes a call to `run()`.



```
public class RunnableInheritingThread implements Runnable {
```

```
    Thread thread;
```

```
    RunnableInheritingThread() {
```

```
        // Create a new, second thread
```

```
        thread = new Thread(this, "Demo Thread");
```

```
        System.out.println("Child thread: " + thread);
```

```
        thread.start(); // Start the thread
```

```
    }
```

```
    public void run() {
```

```
        try {
```

```
            for (int i = 5; i > 0; i--) {
```

```
                System.out.println("Child Thread: " + i);
```

```
                Thread.sleep(500);
```

```
            }
```

```
        } catch (InterruptedException e) {
```


```
            System.out.println("Child interrupted.");
```

```
        }
```

```
        System.out.println("Exiting child thread.");
```

```
    }
```

```
}
```




```
public class RunnableMainApplication {  
    public static void main(String args[]) {  
        new RunnableInheritingThread(); // create a new thread  
        try {  
            for (int i = 5; i > 0; i--) {  
                System.out.println("Main Thread: " + i);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Main thread interrupted.");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```



Creating Multiple Threads

```
public class MultipleThread extends Thread {  
    String name; // name of thread  
  
    MultipleThread(String threadname) {  
        super(threadname);  
        name = threadname;  
        System.out.println("New thread: " + this.getName());  
        this.start(); // Start the thread  
    }  
  
    // This is the entry point for thread.  
    @Override  
    public void run() {  
        try {  
            for (int i = 5; i > 0; i--) {  
                System.out.println(name + ": " + i);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println(name + "Interrupted");  
        }  
        System.out.println(name + " exiting.");  
    }  
}
```



```
public class MultipleThreadMainApplication {  
  
    public static void main(String[] args) {  
        new MultipleThread("One"); // start threads  
        new MultipleThread("Two");  
        new MultipleThread("Three");  
        try {  
            // wait for other threads to end  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            System.out.println("Main thread Interrupted");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```



isAlive() in Thread

The `isAlive()` method tests if this thread is alive. A thread is alive if it has been started and has not yet died.

```
ThreadJoin t1 = new ThreadJoin( threadName: "first thread");  
t1.start();  
t1.isAlive();
```




Joining Threads in Java


`java.lang.Thread` class provides the `join()` method which allows one thread to wait until another thread completes its execution.

If `t` is a `Thread` object whose thread is currently executing, then `t.join()` will make sure that `t` is terminated before the next instruction is executed by the program.

If there are multiple threads calling the `join()` methods that means overloading on join allows the programmer to specify a waiting period. However, as with `sleep`, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.



```
public class ThreadJoin extends Thread {  
    public ThreadJoin(String threadName) {  
        super(threadName);  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 2; i++) {  
            try {  
                Thread.sleep(500);  
                System.out.println("Current Thread: "  
                    + Thread.currentThread().getName());  
            } catch (Exception ex) {  
                System.out.println("Exception has" +  
                    " been caught" + ex);  
            }  
            System.out.println(i);  
        }  
    }  
}
```



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

        ThreadJoin t1 = new ThreadJoin( threadName: "first thread");
        ThreadJoin t2 = new ThreadJoin( threadName: "second thread");
        ThreadJoin t3 = new ThreadJoin( threadName: "third thread");

        // thread t1 starts
        t1.start();

        // starts second thread after when
        // first thread t1 has died.
        try {
            System.out.println("Current Thread: "
                               + Thread.currentThread().getName());
            System.out.println("Thread " + t1.getName() + " is alive: " + t1.isAlive());
            t1.join();
        } catch (Exception ex) {
            System.out.println("Exception has " +
                               "been caught" + ex);
        }

        // t2 starts
        t2.start();

        // starts t3 after when thread t2 has died.
        try {
            System.out.println("Current Thread: "
                               + Thread.currentThread().getName());
            t2.join();
            System.out.println("Thread " + t2.getName() + " is alive: " + t2.isAlive());
        } catch (Exception ex) {
            System.out.println("Exception has been" +
                               " caught" + ex);
        }

        t3.start();
    }
}
```




Thread Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.


The process by which this is achieved is called synchronization.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks.


Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object.

All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.


All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.



```
public class CallMe {  
    void call(String msg) {  
        try {  
            System.out.print("[ " + msg);  
            Thread.sleep( millis: 3000);  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}
```



```
public class Caller implements Runnable {  
    private String msg;  
    private CallMe target;  
    Thread t;  
  
    public Caller(CallMe target, String s) {  
        this.target = target;  
        msg = s;  
        t = new Thread( target: this);  
        t.start();  
    }  
  
    public void run() {  
        target.call(msg);  
    }  
}
```



```
public class SynchronizationMainApplication {  
    public static void main(String args[]) {  
        CallMe target = new CallMe();  
        Caller ob1 = new Caller(target, S: "Hello");  
        Caller ob2 = new Caller(target, S: "Synchronized");  
        Caller ob3 = new Caller(target, S: "World");  
  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```



Output:

```
[Hello[Synchronized[World]  
]  
]
```



Note*: Since the Call method is not synchronized more than one thread can access this method simultaneously and result in the Race Condition.

To prevent from the race condition precede the call() with the keyword synchronized and analyze the output.

```
public class CallMe {  
    synchronized void call(String msg) {  
        try {  
            System.out.print "[" + msg);  
            Thread.sleep( millis: 3000);  
        } catch(InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}
```



Output:

```
[Hello]  
[World]  
[Synchronized]
```



Wait and Notify

wait(): method is invoked to suspend the execution of the thread. The `java.lang.Object.wait()` causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. In other words, this method behaves exactly as if it simply performs the call `wait(0)`.

notify(): method is invoked to wake up the thread. The `java.lang.Object.notify()` wakes up a single thread that is waiting on this object's monitor. If many threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.



JAVA I/O STREAMS

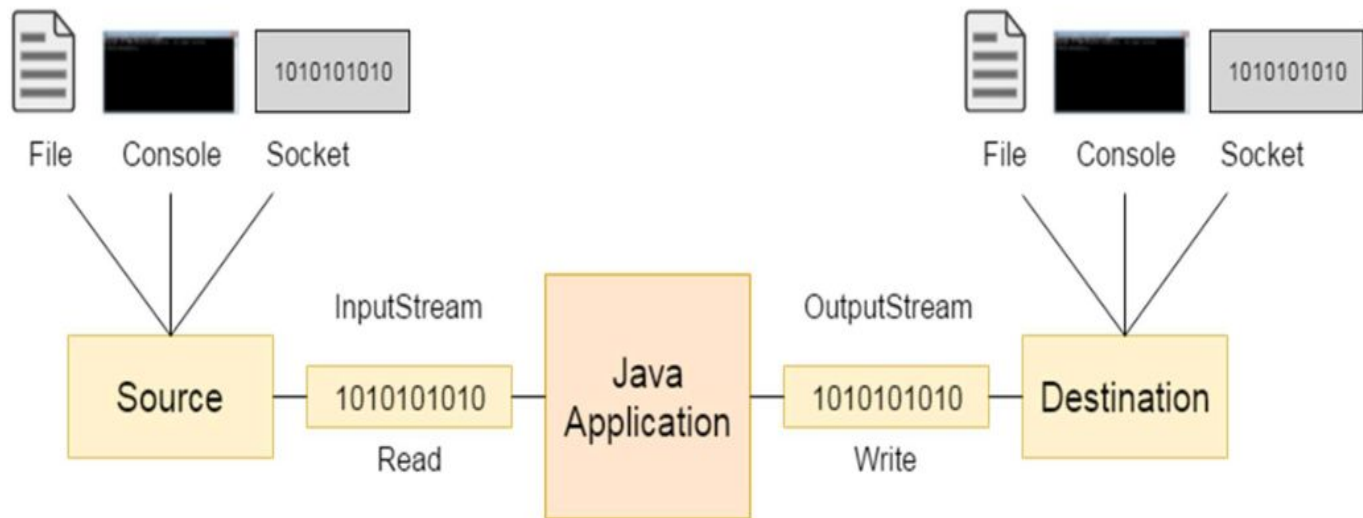
Stream: A stream is a sequence of data. In Java a stream is composed of bytes. A stream is a way of sequentially accessing a file. It's called a stream because it is like a stream of water that continues to flow. There are two kinds of Streams :

1. **OutputStream:**

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

2. **InputStream:**

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.





Character Streams versus Byte Streams

Prior to JDK 1.1, the input and output classes (mostly found in the `java.io` package) only supported 8-bit byte streams.

The concept of 16-bit Unicode character streams was introduced in JDK 1.1.

While byte streams were supported via the `java.io.InputStream` and `java.io.OutputStream` classes and their subclasses, character streams are implemented by the `java.io.Reader` and `java.io.Writer` classes and their subclasses.

Most of the functionality available for byte streams is also provided for character streams.



The methods for character streams generally accept parameters of data type `char` parameters, while byte streams, you guessed it, work with byte data types.

The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix `Reader` or `Writer` and byte-stream classes end with the suffix `InputStream` and `OutputStream`. For example, to read files using character streams, you would use the `java.io.FileReader` class; for reading it using byte streams you would use `java.io.FileInputStream`.

Unless you are working with binary data, such as image and sound files, you should use readers and writers (character streams) to read and write information for the following reasons:

1. They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
2. They are easier to internationalize because they are not dependent upon a specific character encoding.
3. They use buffering techniques internally and are therefore potentially much more efficient than byte streams.



Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. A byte stream access the file byte by byte.

A byte stream is suitable for any kind of file, however not quite appropriate for text files.

There are many classes related to byte streams but the most frequently used classes are, `FileInputStream` and `FileOutputStream`.

FileOutputStream



FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Method	Description
protected void finalize ()	It is used to clean up the connection with the file output stream. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
void write (byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write (byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.



<code>void write(int b)</code>	It is used to write the specified byte to the file output stream.
<code>FileChannel getChannel()</code>	It is used to return the file channel object associated with the file output stream.
<code>FileDescriptor getFD()</code>	It is used to return the file descriptor associated with the stream.
<code>void close()</code>	It is used to closes the file output stream.




```
try {
    FileOutputStream fileOutputStream = new FileOutputStream( name: "testout.txt");
    String s = "Welcome to file handling.";
    byte b[] = s.getBytes();//converting string into byte array
    fileOutputStream.write(b);
    fileOutputStream.close();
    System.out.println("success...");
} catch (Exception e) {
    System.out.println(e);
}
```





FileInputStream Class

FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.



<code>int available()</code>	It is used to return the estimated number of bytes that can be read from the input stream.
<code>int read()</code>	It is used to read the byte of data from the input stream.
<code>int read(byte[] b)</code>	It is used to read up to b.length bytes of data from the input stream.
<code>int read(byte[] b, int off, int len)</code>	It is used to read up to len bytes of data from the input stream.
<code>long skip(long x)</code>	It is used to skip over and discards x bytes of data from the input stream.
<code>FileChannel getChannel()</code>	It is used to return the unique FileChannel object associated with the file input stream.
<code>FileDescriptor getFD()</code>	It is used to return the FileDescriptor object.
<code>protected void finalize()</code>	It is used to ensure that the close method is call when there is no more reference to the file input stream.
<code>void close()</code>	It is used to closes the stream.



```
try {
    FileInputStream fin = new FileInputStream( name: "testout.txt");
    int i = 0;
    while ((i = fin.read()) != -1) {
        System.out.print((char) i);
    }
    fin.close();
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
} catch (IOException e) {
    System.out.println("Some problem occurred while reading " +
        "the file.");
}
```



The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.

Since one of the main purposes of Java is to support the “write once, run anywhere” philosophy, it was necessary to include direct I/O support for characters.

At the top of the character stream hierarchies are the Reader and Writer abstract classes.



Reader

Reader is an abstract class that defines Java's model of streaming character input. All of the methods in this class will throw an `IOException` on error conditions.

Methods	Description
<code>int read()</code>	This method reads a characters from the input stream.
<code>int read(char[] ch)</code>	This method reads a chunk of characters from the input stream and store them in its char array, ch. Returns -1 end of file encountered.
<code>close()</code>	This method closes this output stream and also frees any system resources connected with it.

FileReader:



The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

1. `FileReader(String filePath)`
2. `FileReader(File fileObj)`

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```
public class MainApplication {
    public static void main(String args[]) throws Exception {
        FileReader fr = new FileReader(fileName: "demo.txt");
        BufferedReader br = new BufferedReader(fr);
        String s;
        while((s = br.readLine()) != null) {
            System.out.println(s);
        }
        fr.close();
    }
}
```



Writer

Writer is an abstract class that defines streaming character output.

All of the methods in this class return a void value and throw an IOException in the case of errors.


Methods	Description
<code>abstract void flush()</code>	This method flushes the output steam by forcing out buffered bytes to be written out.
<code>void write(int c)</code>	This method writes a characters(contained in an int) to the output stream.
<code>void write(char[] arr)</code>	This method writes a whole char array(arr) to the output stream.
<code>abstract void close()</code>	This method closes this output stream and also frees any resources connected with this output stream.



FileWriter

FileWriter creates a Writer that you can use to write to a file. Its most commonly used constructors are shown here:

1. `FileWriter(String filePath)`
2. `FileWriter(String filePath, boolean append)`
3. `FileWriter(File fileObj)`
4. `FileWriter(File fileObj, boolean append)`



```
import java.io.FileWriter;

public class MainApplication {
    public static void main(String args[]) throws Exception {
        String source = "Now is the time for all good men\n"
            + "  to come to the aid of their country\n"
            + "  and pay their due taxes.";
        char buffer[] = new char[source.length()];

        /* Copies characters from this string into the destination character array. */
        source.getChars( srcBegin: 0, source.length(), buffer, dstBegin: 0);

        FileWriter f0 = new FileWriter( fileName: "file1.txt");
        for (int i = 0; i < buffer.length; i++) {
            f0.write(buffer[i]);
        }
        f0.close();
        FileWriter f1 = new FileWriter( fileName: "file2.txt");
        f1.write(buffer);
        f1.close();
        FileWriter f2 = new FileWriter( fileName: "file3.txt");
        f2.write(buffer, off: buffer.length - buffer.length / 4, len: buffer.length / 4);
        f2.close();
    }
}
```



Collection in Java

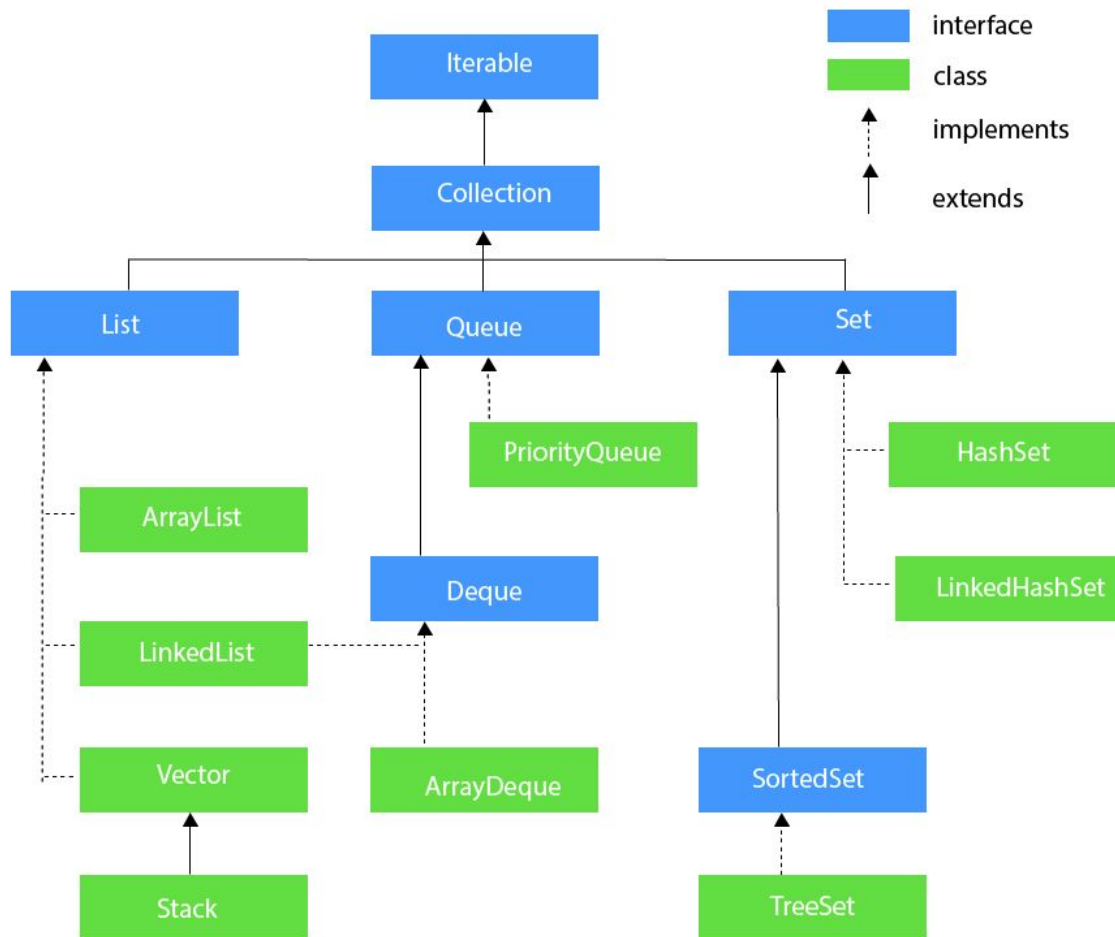
1. The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
2. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
3. Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet, HashMap, SortedMap).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

1. It provides readymade architecture.
2. It represents a set of classes and interfaces.
3. It is optional.





Collection Interface

1. The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
2. Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

1. List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
2. List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
3. To instantiate the List interface, we must use :



```
List <data-type> list1= new ArrayList();
```

```
List <data-type> list2 = new LinkedList();
```

```
List <data-type> list3 = new Vector();
```

```
List <data-type> list4 = new Stack();
```


There are various methods in List interface that can be used to insert, delete, and access the elements from the list.



ArrayList

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.



```
import java.util.ArrayList;
```

```
import java.util.Iterator;
```

```
public class MainApplication {
```

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

```
        ArrayList<String> list = new ArrayList<String>();//Creating arraylist
```

```
        list.add("Ravi");//Adding object in arraylist
```

```
        list.add("Vijay");
```

```
        list.add("Ravi");
```

```
        list.add("Ajay");
```

```
//Traversing list through Iterator
```

```
        Iterator itr = list.iterator();
```

```
        while (itr.hasNext()) {
```

```
            System.out.println(itr.next());
```

```
        }
```

```
    }
```

```
}
```



LinkedList


LinkedList implements the Collection interface.

It uses a doubly linked list internally to store the elements.

It can store the duplicate elements. It maintains the insertion order and is not synchronized.

In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.



```
public class MainApplication {  
    public static void main(String args[]) {  
        LinkedList<String> al = new LinkedList<String>();  
        al.add("Ravi");  
        al.add("Vijay");  
        al.add("Ravi");  
        al.add("Ajay");  
        Iterator<String> itr = al.iterator();  
        while (itr.hasNext()) {  
            System.out.println(itr.next());  
        }  
    }  
}
```




Vector

Vector uses a dynamic array to store the data elements.

It is similar to ArrayList.

However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.



```
import java.util.Iterator;
```

```
import java.util.Vector;
```

```
public class MainApplication {
```

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

```
        Vector<String> v = new Vector<String>();
```

```
        v.add("Ayush");
```

```
        v.add("Amit");
```

```
        v.add("Ashish");
```

```
        v.add("Garima");
```

```
        Iterator<String> itr = v.iterator();
```

```
        while (itr.hasNext()) {
```

```
            System.out.println(itr.next());
```

```
        }
```

```
    }
```

```
}
```




Stack

The stack is the subclass of Vector.

It implements the last-in-first-out data structure, i.e., Stack.

The stack contains all of the methods of Vector class and also provides its methods like `boolean push()`, `boolean peek()`, `boolean push(object o)`, which defines its properties.

Consider the following example.



```
import java.util.Iterator;  
import java.util.Stack;
```

```
public class MainApplication {  
    public static void main(String args[]) {  
        Stack<String> stack = new Stack<String>();  
        stack.push( item: "Ayush");  
        stack.push( item: "Garvit");  
        stack.push( item: "Amit");  
        stack.push( item: "Ashish");  
        stack.push( item: "Garima");  
  
        //Removing entry from stack. Here it removes Garima (LIFO structure)  
        System.out.println(stack.pop());  
  
        Iterator<String> itr = stack.iterator();  
        while (itr.hasNext()) {  
            System.out.println(itr.next());  
        }  
    }  
}
```




Queue Interface

Queue interface maintains the first-in-first-out order.

It can be defined as an ordered list that is used to hold the elements which are about to be processed.

There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:



```
Queue<String> q1 = new PriorityQueue();
```

```
Queue<String> q2 = new ArrayDeque();
```




PriorityQueue

The PriorityQueue class implements the Queue interface.

It holds the elements or objects which are to be processed by their priorities.

PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example



```
import java.util.Iterator;
import java.util.PriorityQueue;

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

        PriorityQueue<String> queue = new PriorityQueue<String>();
        queue.add("Aakash Shaky");
        queue.add("Amit Sharma");
        queue.add("Jai Shankar");
        queue.add("Prarup Gurung");
        queue.add("Raj");

        System.out.println("iterating the queue elements:");

        Iterator itr = queue.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }

        queue.remove();
        queue.poll();

        System.out.println("after removing two elements:");
        Iterator<String> itr2 = queue.iterator();
        while (itr2.hasNext()) {
            System.out.println(itr2.next());
        }
    }
}
```



Deque Interface

Deque interface extends the Queue interface.

In Deque, we can remove and add the elements from both the side.

Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
Deque d = new ArrayDeque( );
```



ArrayDeque


ArrayDeque class implements the Deque interface.

It facilitates us to use the Deque.

Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.



```
import java.util.*;
```

```
public class MainApplication {  
    public static void main(String args[]) {  
  
        //Creating Deque and adding elements  
        Deque<String> deque = new ArrayDeque<String>();  
        deque.add("Gautam");  
        deque.add("Karan");  
        deque.add("Ajay");  
  
        //Traversing elements  
        for (String str : deque) {  
            System.out.println(str);  
        }  
    }  
}
```



Set Interface

Set Interface in Java is present in `java.util` package.

It extends the `Collection` interface.

It represents the unordered set of elements which doesn't allow us to store the duplicate items.

We can store at most one null value in Set. Set is implemented by `HashSet`, `LinkedHashSet`, and `TreeSet`.

Set can be instantiated as:



```
Set<data-type> s1 = new HashSet<data-type>();
```

```
Set<data-type> s2 = new LinkedHashSet<data-type>();
```

```
Set<data-type> s3 = new TreeSet<data-type>();
```



HashSet


HashSet class implements Set Interface.

It represents the collection that uses a hash table for storage.

Hashing is used to store the elements in the HashSet.

It contains unique items.

Consider the following example.



```
import java.util.HashSet;
import java.util.Iterator;

public class SetMainApplication {
    public static void main(String[] args) {
        //Creating HashSet and adding elements
        HashSet<String> set = new HashSet<String>();
        set.add("Ajay");
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ravi");
        //Traversing elements
        Iterator<String> itr = set.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```




LinkedHashSet


LinkedHashSet class represents the LinkedList implementation of Set Interface.

It extends the HashSet class and implements Set interface.

Like HashSet, It also contains unique elements.

It maintains the insertion order and permits null elements.

Consider the following example.



```
import java.util.*;
```

```
public class SetMainApplication {  
    public static void main(String[] args) {  
        HashSet<String> set=new HashSet<>();  
        set.add("Ravi");  
        set.add("Vijay");  
        set.add("Ravi");  
        set.add("Ajay");  
        Iterator<String> itr=set.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next());  
        }  
    }  
}
```



SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements.

The elements of the SortedSet are arranged in the increasing (ascending) order.

The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
SortedSet<data-type> set = new TreeSet();
```



TreeSet


Java TreeSet class implements the Set interface that uses a tree for storage.

Like HashSet, TreeSet also contains unique elements.

However, the access and retrieval time of TreeSet is quite fast.

The elements in TreeSet stored in ascending order.

Consider the following example:



```
import java.util.*;
```

```
public class SetMainApplication {  
    public static void main(String[] args) {  
        TreeSet<String> treeSet=new TreeSet<String>();  
        treeSet.add("Ravi");  
        treeSet.add("Vijay");  
        treeSet.add("Ravi");  
        treeSet.add("Ajay");  
        treeSet.add("Aakash");  
        treeSet.add("Aasha");  
        treeSet.add("123Ahs");  
        //traversing elements  
        Iterator<String> treeItr = treeSet.iterator();  
        while(treeItr.hasNext()){  
            System.out.println(treeItr.next());  
        }  
    }  
}
```



Map interface

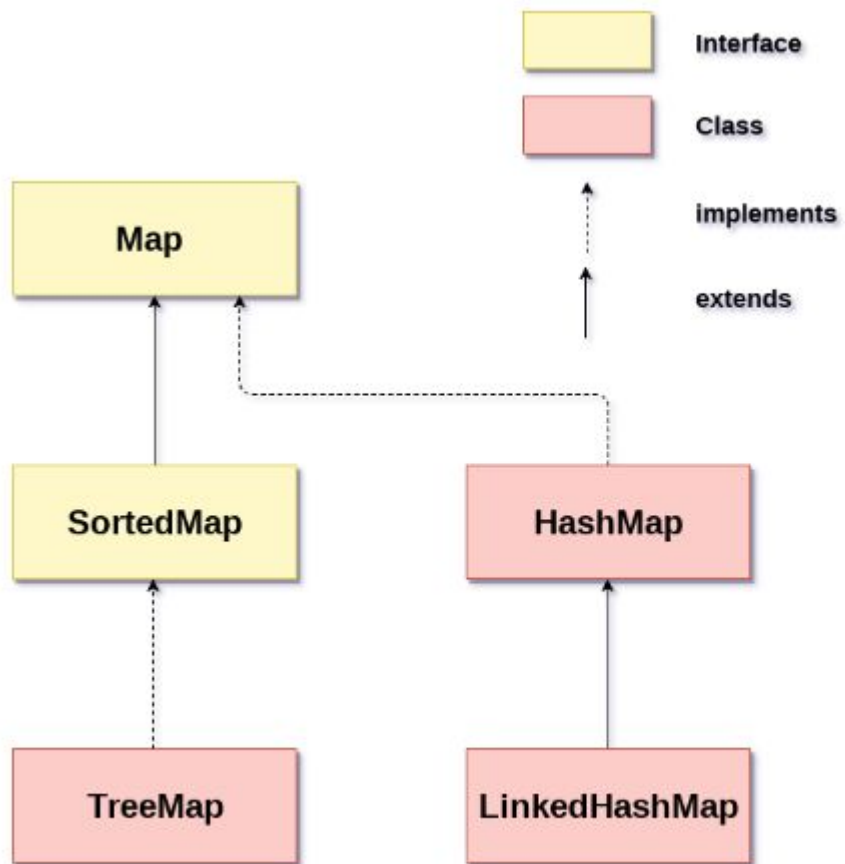
A map contains values on the basis of key, i.e. key and value pair.

Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Map can be instantiated as:

```
Map<data-type, data-type> map = new HashMap<>();
```






HashMap

Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

Points to remember:

1. Java HashMap class contains values based on the key.
2. Java HashMap class contains only unique keys.
3. Java HashMap class may have one null key and multiple null values.
4. Java HashMap class is non synchronized.
5. Java HashMap class maintains no order.



```
import java.util.HashMap;  
import java.util.Map;
```

```
public class MapMainApplication {  
    public static void main(String[] args) {  
        HashMap<Integer, String> hm = new HashMap<Integer, String>();  
        System.out.println("Initial list of elements: " + hm);  
        hm.put(100, "Aakash");  
        hm.put(101, "Prarup");  
        hm.put(102, "Sujan");  
  
        System.out.println("After invoking put() method ");  
        for (Map.Entry m : hm.entrySet()) {  
            System.out.println(m.getKey() + " " + m.getValue());  
        }  
    }  
}
```




LinkedHashMap

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Points to remember:

1. Java LinkedHashMap contains values based on the key.
2. Java LinkedHashMap contains unique elements.
3. Java LinkedHashMap may have one null key and multiple null values.
4. Java LinkedHashMap is non synchronized.
5. Java LinkedHashMap maintains insertion order.



```
import java.util.LinkedHashMap;
```

```
import java.util.Map;
```

```
public class MapMainApplication {
```

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

```
        LinkedHashMap<Integer,String> hm = new LinkedHashMap<Integer,String>();
```

```
        hm.put(101,"Vijay");
```

```
        hm.put(100,"Amit");
```

```
        hm.put(102,"Rahul");
```

```
        for(Map.Entry m : hm.entrySet()){
```

```
            System.out.println(m.getKey() + " " + m.getValue());
```

```
        }
```

```
    }
```

```
}
```






TreeMap

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

1. Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
2. Java TreeMap contains only unique elements.
3. Java TreeMap cannot have a null key but can have multiple null values.
4. Java TreeMap is non synchronized.
5. Java TreeMap maintains ascending order.



```
import java.util.Map;
import java.util.TreeMap;

public class MapMainApplication {
    public static void main(String[] args) {
        TreeMap<Integer, String> map = new TreeMap<Integer, String>();
        map.put(100, "Amit");
        map.put(102, "Ravi");
        map.put(101, "Vijay");
        map.put(103, "Rahul");

        for (Map.Entry m : map.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
        }
    }
}
```