**Java**

1. Object Oriented Programming Language.
2. Platform independent, Simple, multithreaded and Architectural neutral.
3. **Interpreted Language:** Once it is compiled it converted to the byte code which can run in any of the OS.
4. Used to create Enterprise Application.

Sample Java program:

public class MyFirstJavaProgram

{

public static void main (String [] args)

{

System.out.println("Hello World");

}

}

**Object:**

1. Instance of a class
2. it is created from a class
3. In java new keyword is to create new objects.

**Class**

1. User defined data type with data members and member functions.
2. A class is a blue print from which individual objects are created.
3. When a class is defined, no memory is allocated but when it is instantiated (Objects is created) then the memory is allocated.

A sample of class:

public class Dog

{

String breed;

int age;

String color;

void barking ()

{

}

void hungry ()

{

}

void sleeping ()

{

}

}

**Variables**

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

Example: Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to this method only.

public class Test

{

public void pupAge ()

{

int age = 0;

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[])

{

Test test = new Test();

test.pupAge();

}

}

1. **Instance variables:** variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
2. **Class variables:** variables declared within a class, outside any method, with the static keyword.

**Modifiers**

1. Access Modifiers – Default, public, private, protected
2. Non-Access Modifiers – final, abstract, strictfp

**Default:** When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default.

**Private:** The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared. Any other class ofthe same package will not be able to access these members.

private means “only visible within the enclosing class”.

**Protected:** protected means “only visible within the enclosing class and any subclasses”. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

**Public:** Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

**Return Type:**

The type of the value that return the function or method.

**Arrays**

Objects that store multiple variables of the same type. Give the size of the array while declaring. Array index starts from 0. But array length gives the appropriate size of the array.

**Enums**

Restrict a variable to have one of only a few predefined values. The values in this enumerated list are called Enums.E.g.:

For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium and large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.Sample Program:

Class FreshJuice

{

enum FreshJuiceSize{ SMALL, MEDUIM, LARGE}

FreshJuiceSize size;

}

public class FreshJuiceTest

{

public static void main (String args [])

{

FreshJuice juice =new FreshJuice ();

juice.size =FreshJuice.FreshJuiceSize.MEDUIM ;

}

}

**Constructors**

1. Every class has a constructor.
2. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.
3. Each time a new object is created, at least one constructor will be invoked.
4. The main rule of constructors is that they should have the same name as the class.
5. A class can have more than one constructor.

Sample Constructor Program

public class Abc

{

public Abc ()

{

}

public Abc (String name)

{

}

}

**Data Types**

1. **Primitive Data Types:**

**Byte:** 8-bit signed two's complement integer. Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.Example: byte a = 100.

**Short:** 16-bit signed two's complement integer. Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int.

Example: short s= 10000.

**Int:** 32-bit signed two's complement integer. Int is generally used as the default data type for integral values unless there is a concern about memory.

Example: int a = 100000

**Long:**  64-bit signed two's complement integer. This type is used when a wider range than int is needed.Example: int a = 100000L

**Float:** single-precision 32-bit IEEE 754 floating point. Float is mainly used to save memory in large arrays of floating-point numbers. Float data type is never used for precise values such as currency.Example: float f1 = 234.5f

**Double:** double-precision 64-bit IEEE 754 floating point. generally used as the default data type for decimal values, generally the default choice. Double data type should never be used for precise values such as currency.

Example: double d1 = 123.4

**Boolean:** Represents one bit of information. Only two possible values: true and false. Used for simple flags that track true/false conditions.

Example: boolean one = true**Char:** Single 16-bit Unicode character. Char data type is used to store any character.

Example: char letterA ='A'

1. **Reference/Object Data Types:**
2. Reference variables are created using defined constructors of the classes.
3. They are used to access objects.
4. These variables are declared to be of a specific type that cannot be changed.
5. Class objects and various types of array variables come under reference data type.
6. Example: Animal animal = new Animal("giraffe");

**Operators**

Arithmetic Operators, Relational Operators, Bitwise Operators, Logical Operators, Assignment Operators, Misc Operators.

**Loop Controls**

1. **while Loop**

while (Boolean expression)

{

Statements

}

Example;

public class Test

{

public static void main (String args [])

{

int x =10;

while( x <20)

{

System.out.print("value of x : "+ x );

x++;

System.out.print("\n");

}

}

}

1. **do...while Loop**

do

{

Statements

} while (Boolean expression);

Example;

public class Test

{

public static void main (String args [])

{

int x =10;

do

{

System.out.print("value of x : "+ x );

x++;

System.out.print("\n");

} while (x <20);

}

}

1. **for Loop**

for (initialization; Boolean expression; update)

{

Statements

}

Example:

public class Test

{

public static void main (String args [])

{

For (int x =10; x <20; x = x+1)

{

System.out.print("value of x: "+ x);

System.out.print("\n");

}

}

}

**Break**

1. The break keyword is used to stop the entire loop.
2. The break keyword must be used inside any loop or a switch statement.
3. The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.
4. Syntax: break;

Example;

public class Test

{

public static void main (String args [])

{

int [] numbers = {10,20,30,40,50};

for (int x: numbers)

{

if (x ==30)

{

break;

}

System.out.print(x);

System.out.print("\n");

}

}

}

**Continue**

1. The continue keyword can be used in any of the loop control structures.
2. It causes the loop to immediately jump to the next iteration of the loop.
3. In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
4. In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.
5. Syntax: Continue;

Example;

public class Test

{

public static void main(String args[])

{

int[] numbers ={10,20,30,40,50};

for(int x : numbers)

{

if( x ==30)

{

continue;

}

System.out.print(x);

System.out.print("\n");

}

}

}

**Decision Making Statements**

1. **if statements**

An if statement consists of a Boolean expression followed by one or more statements**.**

If (Boolean expression)

{

Statements will execute if the Boolean expression is true

}

E.g.;

public class Test

{

public static void main (String args [])

{

int x =10;

if (x <20)

{

System.out.print("This is if statement");

}

}

}

**If…else Statements**

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

If (Boolean expression)

{

Executes when the Boolean expression is true

}

Else

{

Executes when the Boolean expression is false

}

Example:

public class Test

{

public static void main (String args [])

{

int x =30;

if (x <20)

{

System.out.print("This is if statement");

}

Else

{

System.out.print("This is else statement");

}

}

}

**if...else if...else Statement**

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

If (Boolean\_expression1)

{

//Executes when the Boolean expression 1 is true

}

Elseif (Boolean\_expression2)

{

//Executes when the Boolean expression 2 is true

}

elseif (Boolean\_expression3)

{

//Executes when the Boolean expression 3 is true

}

Else

{

//Executes when the none of the above condition is true.

}

Example:

public class Test

{

public static void main (String args [])

{

int x =30;

if (x ==10)

{

System.out.print("Value of X is 10");

}

elseif (x ==20)

{

System.out.print("Value of X is 20");

}

elseif (x ==30)

{

System.out.print("Value of X is 30");

}

Else

{

System.out.print("This is else statement");

}

}

}

**Nested if...else Statement**

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

if(Boolean\_expression1)

{

//Executes when the Boolean expression 1 is true

if(Boolean\_expression2)

{

//Executes when the Boolean expression 2 is true

}

}

Example

public class Test

{

public static void main(String args[])

{

int x =30;

int y =10;

if( x ==30)

{

if( y ==10)

{

System.out.print("X = 30 and Y = 10");

}

}

}

1. **Switch Statements**
2. A switch statement allows a variable to be tested for equality against a list of values.
3. Each value is called a case, and the variable being switched on is checked for each case.

switch(expression)

{

case value :

//Statements

break;//optional

case value :

//Statements

break;//optional

//You can have any number of case statements.

default://Optional

//Statements

}

Example:

public class Test

{

public static void main(String args[])

{

char grade = args[0].charAt(0);

switch(grade)

{

case'A':

System.out.println("Excellent!");

break;

case'B':

case'C':

System.out.println("Well done");

break;

case'D':

System.out.println("You passed");

case'F':

System.out.println("Better try again");

break;

default:

System.out.println("Invalid grade");

}

System.out.println("Your grade is "+ grade);

}

}

**Blocks**

1. Group of one or more statements enclosed in braces.
2. A block begins with an opening brace ({) and ends with a closing brace (}).
3. Between the opening and closing braces, you can code one or more statements.
4. A block is itself a type of statement.

Example

Public static void main (String args [])

{

Int x=1;

System.out.println(x);

{

Int y=0;

System.out.println(y); // y dies here

}

System.out.println(y);

// y is uninitialized inside the block so it won’t access after the block

}

**Pre-Increment and Post-Increment**

1. Pre-increment means that the variable is incremented BEFORE it's evaluated in the expression.
2. Post-increment means that the variable is incremented AFTER it has been evaluated for use in the expression.

Example:

Int x=0;

Int y=1;

System.out.println(x++); // Output will be 0

System.out.println(++x); // Output will be 1

System.out.println(y--); // Output will be 1

System.out.println(--y); // Output will be 0

**OOP Concepts**

**Inheritance**

1. Mechanism in which one object acquires all the properties and behaviors of a parent object.
2. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).
3. Inheritance reduced redundancy
4. Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
5. Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
6. reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class.

class Subclass-name extends Superclass-name

{

    //methods and fields

}

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

class Employee

{

  float salary=40000;

}

class Programmer extends Employee

{

  int bonus=10000;

  public static void main (String args [])

{

   Programmer p=new Programmer ();

   System.out.println("Programmer salary is:"+p.salary);

    System.out.println("Bonus of Programmer is:"+p.bonus);

}

}

**Polymorphism**

Concept by which we can perform a single action in different ways. The word "poly" means many and "morphs" means forms. So, polymorphism means many forms. Two types of polymorphism in Java: **compile-time polymorphism and runtime polymorphism.** We can perform polymorphism in java by method overloading and method overriding.

Runtime polymorphism

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

**Upcasting:** If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

Example of upcasting:

class A

{

}

class B extends A

{

}

A a=new B (); //upcasting

class Bike

{

   void run(){System.out.println("running");}

}

class Splendor extends Bike

{

  void run()

{

System.out.println("running safely with 60km");

}

  public static void main(String args[])

{

    Bike b = new Splendor(); //upcasting

    b.run();

  }

}

Here we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

[**Method Overriding**](https://www.geeksforgeeks.org/overriding-in-java/) **:** Occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example:

class Parent

{

    void Print()

    {

        System.out.println("parent class");

    }

}

class subclass1 extends Parent

{

  void Print()

     {

         System.out.println("subclass1");

     }

}

class subclass2 extends Parent

{

   void Print()

     {

         System.out.println("subclass2");

     }

}

class TestPolymorphism3

{

    public static void main(String[] args)

    {

 Parent a;

a = new subclass1();

         a.Print();

   a = new subclass2();

         a.Print();

    }

}

Compile Time Polymorphism

Also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java doesn’t support the Operator Overloading**.

**Method Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments**or/and**change in type of arguments.**

Example:

class MultiplyFun

{

static int Multiply(int a, int b)

     {

         return a \* b;

     }

       static double Multiply(double a, double b)

     {

         return a \* b;

     }

}

class Main

{

    public static void main(String[] args)

    {

System.out.println(MultiplyFun.Multiply(2, 4));

System.out.println(MultiplyFun.Multiply(5.5, 6.3));

    }

}

**Abstraction**

1. Only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user.
2. The process of identifying only the required characteristics of an object ignoring the irrelevant details.
3. In java, abstraction is achieved by [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) and [abstract classes](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.

Abstract Classes and Abstract Methods:

1. An abstract class is a class that is declared with an [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method defined abstract must always be redefined in the subclass, thus making [overriding](https://www.geeksforgeeks.org/overriding-in-java/) compulsory OR either make the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the [new operator](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

**Encapsulation**

Wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield. Encapsulated code is easy to test for unit testing.

**Super Keyword**

1. Reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance.
2. **Use of Super keyword with Variables:** This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM.

Example:

class Vehicle

{

    int maxSpeed = 120;

}

class Car extends Vehicle

{

int maxSpeed = 180;

void display()

     {

        System.out.println("Maximum Speed: " + super.maxSpeed);

     }

}

class Test

{

public static void main(String[] args)

     {

         Car small = new Car();

         small.display();

     }

}

1. **Use of Super Keyword with Methods:** when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

class Person

{

    void message ()

    {

        System.out.println("This is person class");

    }

}

class Student extends Person

{

    void message ()

    {

        System.out.println("This is student class");

    }

    void display ()

    {

        // will invoke or call current class message () method

        message ();

        // will invoke or call parent class message () method

        super.message();

    }

}

  class Test

{

    public static void main (String args [])

    {

        Student s = new Student ();

        // calling display () of Student

        s.display();

    }

}

1. **Use of Super Constructors:** can also be used to access the parent class constructor. One more important thing is that, super can call both parametric as well as non-parametric constructors depending upon the situation.

class Person

{

    Person ()

    {

        System.out.println("Person class Constructor");

    }

}

  class Student extends Person

{

    Student ()

    {

        // invoke or call parent class constructor

        Super ();

System.out.println("Student class Constructor");

    }

}

  class Test

{

    public static void main (String [] args)

    {

        Student s = new Student ();

    }

}

**This Keyword**

‘this’ is a reference variable that refers to the current object.

**Using ‘this’ keyword to refer current class instance variables:**

class Test

{

    int a;

    int b;

    // Parameterized constructor

    Test (int a, int b)

    {

        this.a = a;

        this.b = b;

    }

 void display()

    {

        System.out.println("a = " + a + "  b = " + b);

    }

  public static void main(String[] args)

    {

        Test object = new Test(10, 20);

        object.display();

    }

}

**Static**

Static is a non-access modifier in Java which is applicable for blocks, variables, methods, nested classes. To create a static member (block, variable, method, nested class), precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

**Interface**

1. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
2. Interfaces specify what a class must do and not how. It is the blueprint of the class.
3. An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So, it specifies a set of methods that the class has to implement.
4. If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

interface <interface name>

{

// declare constant fields

// declare methods that abstract

// by default.

}

interface Player

{

    final int id = 10;

    int move ();

}

**Exception Handling**

Exception is an abnormal condition. It is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. **Checked Exception:** The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.
2. **Unchecked Exception**: The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.
3. **Error:** Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java provides five keywords that are used to handle the exception. The following table describes each.

1. **Try**: The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
2. **Catch**: The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
3. **Finally**: The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
4. **Throw**: The "throw" keyword is used to throw an exception.
5. **Throws**: The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

**File Handling**

A **File** is an abstract data type. A named location used to store related information is known as A **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File.**

**Stream**

A series of data is referred to as **a stream**. In [Java](https://www.javatpoint.com/java-tutorial), **Stream** is classified into two types,

**Byte Stream:** Is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

**Character Stream:** Is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

**File Operations**

We can perform the following operation on a file:

* **Create a File:** **Create a File** operation is performed to create a new file. We use the **createNewFile()** method of file. The **createNewFile()** method returns true when it successfully creates a new file and returns false when the file already exists.
* **Get File Information:** The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.
* **Write to a File:** The next operation which we can perform on a file is **"writing into a file".** In order to write data into a file, we will use the **FileWriter** class and its **write()** method together. We need to close the stream using the **close()** method to retrieve the allocated resources.
* **Read from a File:** The next operation which we can perform on a file is **"read from a file".** In order to write data into a file, we will use the **Scanner** class. Here, we need to close the stream using the **close()**method. We will create an instance of the [Scanner class](https://www.javatpoint.com/Scanner-class) and use the [hasNextLine() method](https://www.javatpoint.com/post/java-scanner-hasnextline-method) [nextLine() method](https://www.javatpoint.com/post/java-scanner-nextline-method) to get data from the file.
* **Delete a File:** The next operation which we can perform on a file is **"deleting a file".** In order to delete a file, we will use the **delete()**method of the file. We don't need to close the stream using the **close()** method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

**Collections**

A framework that provides an architecture to store and manipulate the group of objects. A Collection represents a single unit of objects, i.e., a group.

**Iterator Interface**

Iterator interface provides the facility of iterating the elements in a forward direction only.

**Iterable Interface**

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

**Collection Interface**

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.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean add All (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

**List Interface**

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.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

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();

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.

It can be instantiated as:

Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();

1. **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.
2. **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.

It can be instantiated as:

Deque d = new ArrayDeque();

1. **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.
2. **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.
* **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.
* **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.

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.

**Generics**

Generics mean parameterizedtypes. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

There are mainly 3 advantages of generics.

1. **Type-safety:** We can hold only a single type of objects in generics.
2. **Type casting is not required:** There is no need to typecast the object.
3. **Type casting is not required:** There is no need to typecast the object.

**Generic class**

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

Creating a generic class:

class MyGen<T>

{

T obj;

void add(T obj){this.obj=obj;

}

T get()

{

return obj;

}

}

**Type Parameters**

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

**Generic Method**

We can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

**Threads**

Threads allows a program to operate more efficiently by doing multiple things at the same time. Threads can be used to perform complicated tasks in the background without interrupting the main program. There are two ways to create a thread. It can be created by extending the Thread class and overriding its run() method:

**Thread Model**

**1) New (Ready to run)**

A thread is in **New** when it gets CPU time.

**2) Running**

A thread is in **a Running** state when it is under execution.

**3) Suspended**

A thread is in the **Suspended** state when it is temporarily inactive or under execution.

**4) Blocked**

A thread is in the **Blocked** state when it is waiting for resources.

**5) Terminated**

A thread comes in this state when at any given time, it halts its execution immediately.

**Creating Thread**

A thread is created either by "creating or implementing" the **Runnable Interface** or by extending the **Thread class**. These are the only two ways through which we can create a thread.

**Thread Class**

A **Thread class** has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the **Object** class. The **Object** class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

**Thread()**

**Thread(Runnable, String name)**

**Thread (Runnable target)**

**Thread (ThreadGroup group, Runnable target, String name)**

**Thread (ThreadGroup group, Runnable target)**

**Thread (ThreadGroup group, String name)**

**Thread (ThreadGroup group, Runnable target, String name, long stackSize)**

**Runnable Interface(run() method)**

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the **run()** method to perform an action for the thread.

**start() method**

The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the **start()** method, the thread changes the state from New to Runnable. It executes the **run() method** when the thread gets the correct time to execute it.

**Creating thread by implementing the runnable interface**

In Java, we can also create a thread by implementing the runnable interface. The runnable interface provides us both the run() method and the start() method.

**Inner class**

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1. **Nested Inner class**: can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
2. **Method Local inner classes**: Inner class can be declared within a method of an outer class.
3. **Anonymous inner classes**: Anonymous inner classes are declared without any name at all.
4. **Static nested classes:** Static nested classes are not technically an inner class. They are like a static member of outer class.

**Garbage Collection**

Process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory. It is automatically done by the garbage collector (a part of JVM) so we don't need to make extra efforts.