| **Introduction:**  **Need of java:**  **Language:** It is used for communication between two people.  **Programming language:** It is used for communication between humans and systems. Communicating with a system means giving commands using software i.e using software. And to develop that software we required a programming language.  **Need of java:**  Using C language if we develop software that software we can run on the same o.s because C language is platform dependent.  So if we have to develop platform independent software we have to use java.  **Features of java**   1. **Simple:**   Java is simple as a learner because its syntax is the same as C,C++ so learners can easily understand and it is easy as a developer because they have removed pointers so it’s easy to program.    **2. Platform Independent:**  Java is platform independent because compiled code of java we can run on any os. Because compiled code of java does not contain machine instructions, it contains bytecode which does not belong to any os. Java achieves platform independence because of Byte code.  In java byte code is platform independent but JVM is platform dependent i.e each os has its own JVM.  **3. Architectural neutral:**  There are no implementation dependent features e.g. size of primitive types is fixed.  In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.  Hence java is architectural neutral with respect to memory management.  **4. Robust:**  Robust simply means strong. Java uses strong memory management. There is a lack of pointers that avoids security problems. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points make java robust.  **5. Portable:**  We may carry the java bytecode to any platform.  **6. Distributed:**  We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet. **7. Multi-threaded:** A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads.  The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multimedia, Web applications etc.  **8. Secure:**  Java is secure because :   1. Removed pointer 2. Byte code verifier which checks whether the byte code is generated by proper compiler or not. 3. Along with this they have given four specifiers public, private, protected and default. |
| --- |
| **Java software’s:**  There are two types of java software which we have to deploy to work with java.  These are given below:   1. JDK(Java Development Kit) 2. JRE(Java Runtime Environment)    When we have a .class file and we have to execute those .class files then we require only JVM so we can have only JRE. But when we have a .java file then we have to compile first and then we have to execute so in this case we require a compiler plus JVM so we can have a JDK. **First Program**   | public class Demo {  public static void main(String[] args) {  System.out.println("Hello");  }  }  Output: Hello | | --- | |
| **What is class and Object?** If we want to store a student's rollNo and name under one name then there is no predefined data type which can store two different values. So in this case, we have to create our own data type.  | Class is a keyword which is used to develop user defined data type.  Or class is a user defined data type. | | --- |   So using class keyword we will develop new data type student as below:   | public class Student {  int rollNo;  String name;  } | Here Student is a new data type which will store one integer and one string value.  But class is just imagination. It does not have physical existence. If we want to store rollNo and the name of the student we require memory allocation for rollNo and name.  If we want to allocate memory for class variables then we have to create an object of that class.  Creating an object means allocating memory for it’s non-static variables. | | --- | --- |  | **Object** is a physical existence of a class or it is an instance of a class. | | --- |  | **Student.java**  public class Student {  int rollNo;  String name;  public static void main(String[] args) {  // Storing first student info  Student s = new Student();//object creation statement  // storing value  s.rollNo = 101;  s.name = "sam";  // storing second student info  Student s1 = new Student();  s1.rollNo = 102;  s1.name = "ram";  // retriving data  System.out.println(s.rollNo+" "+s.name);  System.out.println(s1.rollNo+" "+s1.name);  }  }  Output:  101 sam  102 ram |  | | --- | --- |   If we want to store student name and college name then we have to create class as given below:   | public class Student {  String student\_name;  String college\_name;  public static void main(String[] args) {  // Storing first student info  Student s = new Student();//object creation statement  // storing value  s.student\_name = "sam";  s.college\_name = "JSPM";  // storing second student info  Student s1 = new Student();  s1.student\_name = "ram";  s1.college\_name = "JSPM";  // retrieving data  System.out.println(s.student\_name+" "+s.college\_name);  System.out.println(s1.student\_name+" "+s1.college\_name);  }  }  Output:  Sam JSPM  Ram JSPM | | --- |   Here in above example college\_name property is common for all objects so declare this property as static as given below:   | **Student.java**  public class Student {  String student\_name;  static String college\_name = "JSPM";  public static void main(String[] args) {  Student s = new Student();  s.student\_name = "sam";  Student s1 = new Student();  s1.student\_name = "ram";  System.out.println(s.student\_name+" "+s.college\_name);  System.out.println(s1.student\_name+" "+s1.college\_name);  }  }  Output:  Sam JSPM  Ram JSPM |  | | --- | --- |   Static variable gets memory after class loading and before main method.  And static variable is accessible to every object and every member of that same class.  When we create an object of a class it allocates memory to only non-static variable not to static variable.  In the above example student\_name is a non-static variable and college\_name is a static variable. |
| **Data Types**  Int a;  Here ‘a’ is a data type.  Data type tells us three things:   1. How many bytes of memory JVM should allocate for that variable a. 2. Which type of value we can store inside that location. 3. Which type of operation we can perform on that data.   In java there are 8 data type given below:    **Example:**   | 1. byte b = 10;   int a = b;  System.out.println(a); | 2) float f = 10.5f;  int a = (int)f;  System.out.println(a); | 3) float f = 10.5f;  float f = (float)10.5; | | --- | --- | --- | | 4) char ch = 'a';  char ch1 = 98; | 5) char ch = ‘9’  Char ch1 = ‘98’ | 6) char ch = 256;  char ch1 = 777; |   Type casting: when we store one type of value into another type we require type casting.  Type casting means converting one type of value into another type.  Example 1.  Here b value will internally convert into int type. JVM will do this because I am storing small values into large one so data will not be lost. Hence JVM will take responsibility and it will perform implicit type casting. It is also called weddening.  Example 2.  Here float is larger than int hence data will lose while storing hence JVM will not take responsibility. He has to convert explicitly. This is called explicit type casting or narrowing.  Example 3.  Here 10.5 is by default double value. So we have to type cast into float.  We can type cast double value into float using two ways as given in example 3.  Example 4.  Here type of ch1 is char so in ch1 variable JVM will store ‘b’ not 98.  98 is an ascii value of ‘b’.  Example 5.  Here if we write char ch = 9, JVM will store it’s respective character but if we want to store 9 as character we have to write inside a single quote.  Char ch1 = ‘98’ here we will get errors because inside a single quote we can write only one character.  Example 6.  Here in ch , JVM will store ‘?’ That is a special character in java. Because the ascii range is from 0 to 255 and if we store any 255 plus value then JVM will store ‘?’.  Inside ch1 also JVM will store ‘?’. |
| **Variable and its types**  Variable is named memory location or name of a reserved memory location which is used to store data temporarily.  For example:  When we declare int a;  Four bytes of memory will be reserved under name a i.e a is the name for that reserved memory area.  There are two types of variables depends on its data type:   1. Normal or primitive variable: The variable whose type is primitive data type is called a primitive variable.   For example:  int a = 10;  Here a primitive variable and primitive variable can store values directly.    2. Reference variable: The variable whose type is reference data type i.e class, interface or enum is called as reference variable.  For example:  Demo d; -- Here d is a reference variable because it’s type is Demo and Demo is a class.  String s; -- Here s is a reference variable because it’s type is String and String is a class.  Reference variable never stores value directly. It stores references to the object which has stored data.  For example:  String s = “sam”;  Here String class objects will create which store data sam. And reference of that object stored into s.    Mainly there are two types of variable:   1. Local variable 2. Class level variable   a. Static variable or class variable  b. Non-static variable or instance or object level variable   1. **Local variable**     The variable which is declared inside a method or block is called a local variable.  Local variables will get memory when we call that method and this memory will get deallocated once method execution completes. I.e lifetime of a local variable is from method call to method execution.  Scope of the local variable is within the same method i.e local to that same method where it declared.   | class Demo {  public static void m1(){  int m = 10;  System.out.println(m);//10  System.out.println(n); //-- 2  }  public static void main(String[] args) {  int n = 20;  m1();  System.out.println(n);//20  System.out.println(m);// -- 1  }  } | Here m is local variable of m1() and  n is the local variable of main() so they can be accessed within the same method.  At line number 1 we will get error because m is local variable of m1()  And we cannot access outside the m1().  Similarly at line 2 we will get error because n is local variable of main()  Cannot access in m1(). | | --- | --- | | class Demo {  public static void main(String[] args) {  int n;  System.out.println(n);  }  } | Here we will get errors because we have to initialize local variables before use. | | class Demo {  public static void main(String[] args) {  int n;  System.out.println("Hello");  }  } | Here we will get the output ‘Hello’.  Here we have not initialized variable n still we will not get error because we have not used variable n.  And the rule is before use we have to initialize. | | class Demo {  public static void main(String[] args) {  static int n = 10;  System.out.println(n);  }  } | Here we will get errors because we cannot declare local variables as static |  | Note: We cannot declare local variables as static because generally we declare a variable as static when that variable is common to all objects and needs to be accessed throughout the class.  But local variables are not available outside the method so there is no sense of declaring local variables as static. | | --- |   Whenever we have to store some temporary results generated in a method, we declare a local variable.  **2. Static variable**  The variable which is declared at class level with a static keyword called static variable.  Static variables will get memory after class loading and before main method invocation.  Static variables are available throughout the class and accessible to all objects of that class.   | Lifetime of static variables is from class loading to class unloading.  Scope of static variables is throughout the class. | | --- |   **Example:**   | class Demo {  static int a = 10;  public static void main(String[] args) {  System.out.println(a);  System.out.println(b);  }  static int b = 20;  } |  | | --- | --- | | class Demo {  static int a = 10;  public static void main(String[] args) {  System.out.println(a);  System.out.println(b);  m1();  }  public static void m1(){  System.out.println(a);  System.out.println(b);  }  static int b = 20;  } | Output :  10  20  10  20 |   **3. NonStatic variable**  The variable declared at class level and without static keyword is called as non static variable.  Non Static variable gets memory on object creation. When we create an object of a class , JVM allocates memory for it’s non static variables.  To access non static variables we require an object. Without objects, non static variables can not exist.   | public class Demo {  int a ;  int b ;  public static void main(String[] args) {  Demo d = new Demo();  d.a = 10;  d.b = 20;  System.out.println(d.a+" "+d.b);  Demo d1 = new Demo();  d1.a = 100;  d1.b = 200;  System.out.println(d1.a+" "+d1.b);  }  } |  | | --- | --- |  | Lifetime of non-static variables is from object creation to object destruction.  Scope is object scope i.e wherever we can access object there we can access non-static variable. | | --- | |
| **Methods and its types**   | public class Demo {  static int a = 10;  a = 20;  System.out.println(a);  public static void main(String[] args) {    }  } | Here we will get errors at line 3 and 4 because we can not write any type of statement at class level except variable declaration and initialization.  We have to write these statements inside the method. | | --- | --- |  | Method is a collection of statements which perform specific tasks. | | --- |   Advantages:   1. Code reusability. 2. Increases code readability. 3. Using this method we can get modularity. 4. Maintenance becomes easy.   **Types of method:**   1. **Parameterized and non parameterized method.**  | public class Demo {  public static void add() {  int c = 10 + 20;  System.out.println(c);  }    public static void main(String[] args) {  System.out.println("main method");  add();  }  } | public class Demo {  public static void add(int m, int n) {  int c = m + n;  System.out.println(c);  }  public static void main(String[] args) {  System.out.println("main method");  add(10,20);  }  } | | --- | --- | | public class Demo {  public static void add(float a) {  System.out.println(a);  }  public static void main(String[] args) {  System.out.println("main method");  add(10.5f);  add(10);  }  } | Float parameterized method means we can pass float or other values which can fit into float i.e byte,short,int,long etc.  So in this example when we pass 10 it will store as 10.0. | | public class Demo {  int a = 10;  public static void m1(Demo d1) {  System.out.println(d1.a);//10  d1.a = 100;  }  public static void main(String[] args) {  Demo d = new Demo();  System.out.println(d.a);//10  m1(d);  System.out.println(d.a);//100  }  } | Here m1() parameter is Demo so we have to pass the Demo object. Here d and d1 both point to the same object. So if anyone changes their value it will affect other reference variables. |   **2. Void and non void method**  If the method has return type void then we cannot return any value from it. We cannot return any value but we can write a return statement.  If the method return type is non void then it returns respective value i.e if return type is float then we can return float and its compatible values i.e byte, short, int, long etc.   | public class Demo {  public static void m1() {  System.out.println("void method");  }  public static void main(String[] args) {  m1();  }  } | public class Demo {  public static void m1() {  System.out.println("void method");  return;  }  public static void main(String[] args) {  m1();  }  } | | --- | --- |   **Different ways to call a non void methods:**   | class Test {  public static float m1(){  return 10.5f;  }  public static void main(String[] args) {  m1();// statement 1  float f = m1();  float x = 10 + m1();  System.out.println(m1());// statement 2  System.out.println(x);  System.out.println(f);  }  } | **Output:**  **10.5**  **20.5**  **10.5**  At statement 1 method will execute but returned value we can not use further in the program.  At statement 2 method will execute and returned value will print but that value we can not use further. | | --- | --- | | **Output?**  class Test {  public static float m1(){  return 10.5f;  }  public static void main(String[] args) {  int a = m1();// statement 1  }  }  **Remove error at statement 1** | **Output?**  class Test {  int a = 10;  public static void m1(Test t1){  System.out.println(t1.a);  t1.a = 200;  }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.a);  m1(t);  System.out.println(t.a);  }  } |   **Different ways to call void methods:**   | class Test {  int a = 10;  public static void m1(){  System.out.println("m1 method");  }  public static void main(String[] args) {  int x = m1();  int y = 10 + m1();  System.out.println(m1());  m1();//statement 1  }  } | Here only statement 1 is valid call to a void method | | --- | --- |   **3. Static and non-static methods**  If we want to call a method without creating an object then declare that method as static otherwise declared as non-static. |
| **Access Specifiers**:  Access specifiers in java defines access permission for a member (class or method or variable or inner class).  **Types of Access Specifiers:**  In java we have four Access Specifiers and they are listed below.  1. Public  2. Private  3. Protected  4. Default (no specifiers)  We look at these Access Specifiers in more detail.    *Public specifiers:*  Public Specifiers achieve the highest level of accessibility. Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.  **Example:**  public class Demo { // public class public x, y, size; // public instance variables }  *Private specifiers:*  Private Specifiers achieves the lowest level of accessibility. Private methods and fields can only be accessed within the same class to which the methods and fields belong. Private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier. Using Private Specifier we can achieve encapsulation and hide data from the outside world.  **Example:**  public class Demo { // public class private double x, y; // private (encapsulated) instance variables  public set(int x, int y) { // setting values of private fields this.x = x; this.y = y; }  public get() { // setting values of private fields return Point(x, y); } }  *Protected specifiers:*  Methods and fields declared as protected can only be accessed by the subclasses in other packages or any class within the package of the protected members' class. The protected access specifier cannot be applied to class and interfaces.  *Default (no specifier):*  When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword. Classes, variables, and methods can be default accessed. Using default specifier we can access class, method, or field which belongs to the same package, but not from outside this package.  **Example:**  class Demo  { int i; (Default) } |
| **Object Oriented Programming**   1. **Data Hiding** 2. **Abstraction** 3. **Encapsulation** 4. **Inheritance** 5. **Polymorphism** 6. **Abstract class** 7. **Interface** 8. **Adapter class** 9. **Marker Interface**   **Data Hiding:**  Hiding internal confidential data from outsiders and provides only when he provides proper authentication is called data hiding.  In java we achieve data hiding using private keywords.  **Abstraction:**  Hiding internal implementation from end users and providing only services is called abstraction.   | Note: In data hiding we hide data and in abstraction we hide internal coding or implementation. | | --- |   **Encapsulation:**  Encapsulation is a process of binding variables and methods into a single unit.  Class is the best example of encapsulation.  **Inheritance:**  Reusing parent class properties into child class is called inheritance.  Consider in a project we have loan model in which there are three classes as given below:   | Class HomeLoan{  10 common methods  } | Class PersonalLoan {  10 common methods  } | Class VehicleLoan{  10 common methods  } | | --- | --- | --- |   Here all classes belong to loan so there should be some common properties. And suppose there are 10 common methods.  But we have to add those 10 methods in all three classes, so code is repeating here.  Also in maintenance if we want to change in one the common method we have to change at three locations so maintenance also becomes complex.  To overcome the above problems we can take separate class Loan and we can write those common methods inside this class. And simply we will extend this Loan class in all three classes as follows:   | Class Loan{  10 common methods  } | Class HomeLoan extends Loan{  } | Class PersonalLoan extends Loan {  } | Class VehicleLoan extends Loan{  } | | --- | --- | --- | --- |   If we extend one class into another class one class becomes parent and another becomes child.  In the above example Loan becomes parent and HomeLoan and the other two become child classes.  And all the members of parent class are available to child class so we don’t have to write it again.  Here code reusing and maintenance also become easy because if we change in parent class method that will reflect automatically in all child classes.   | class Demo{  int a = 10;  public void m1(){  System.out.println("m1 method");  }  }  class Test extends Demo{    public static void main(String[] args) {  Test t = new Test();  System.out.println(t.a);  t.m1();  }  } | Here all the members of Demo class are available to Test class.  When we call t.a ,JVM will search ‘a’ variable in Test class first , if we don't get it , it will search in Demo class because Demo class members are available to Test class.  Similarly When we call t.m1(), Jvm will search m1() first in Test class , if we don't get there , it will call from Demo class. | | --- | --- |  | When we create an object of a class , it allocates memory for the same class as well as it’s parent class non static variables. But the parent class object will not be created. | | --- |  | class Demo{  int a = 10;  }  class Test extends Demo{  int b = 20;  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.a);  System.out.println(t.b);  }  } |  | | --- | --- | | public class Test {  public void m1(){  System.out.println("m1 method");  }  }  public class Demo extends Test{  public void m2(){  System.out.println("m2 method");  }  public static void main(String[] args) {  Test t = new Demo();  t.m1();  t.m2();// statement 1  }  } | We can store child class objects into parent class reference variable.  In this example we have stored Demo class objects into Test class reference.  In this case we can access only parent class members.  Hence we got an error at statement 1. |  | In java Object class is a default parent class for all the classes. | | --- |  | Class Demo{  } | Internally this class will be  Class Demo extends Object{  } | | --- | --- |   **Multiple Inheritance:**  Extending more than one class in one class is called multiple inheritance and java does not support multiple inheritance i.e in java we cannot extend more than one class.   | class Demo extends Test,Test1{} | Error: Multiple inheritance | | --- | --- |   **Why does java not support multiple inheritance?**  Java does not support multiple inheritance because of ambiguity problem i.e if child class is trying to access a member which is present in both parent class then jvm will get confused from where it should access that member.  Consider following case:   | class Test extends Test1{  int a = 20;  }  public class Test1 {  int a = 10;  }  class Demo extends Test,Test1{  public static void main(String[] args) {  Demo d = new Demo();  System.out.println(d.a);  }  } | Here when we call d.a , jvm will search for a variable in Demo class first if he doesn't get in Demo class, then it will search in it’s parent class Test and Test1.  But both the classes contain ‘a’ variable then jvm will get confused.  Hence java does not support multiple inheritance. | | --- | --- |   **Multilevel Inheritance:**   | class Test1 {  int a = 10;  }  class Test extends Test1{  int b = 20;  }  class Demo extends Test{  public static void main(String[] args) {  Demo d = new Demo();  System.out.println(d.b);  System.out.println(d.a);  }  } | Here all the members of Test1 and Test class are available to Demo class. | | --- | --- | |
| **Polymorphism:**  Polymorphism in java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.  There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.  **Overloading:**  If a class has multiple methods having the same name but different in parameters, it is known as Method Overloading.  If we have to perform only one operation, having the same name of the methods increases the readability of the program.  Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.  So, we perform method overloading to figure out the program quickly.   | In method overloading , method signatures should be different. | | --- |  Advantage of method overloading Method overloading *increases the readability of the program.*   | class Demo {  public void add(){  int c = 10 + 20;  System.out.println(c);  }  public void add(int a,int b){  int c = a + b;  System.out.println(c);  }  public void add(float a,int b){  float c = a + b;  System.out.println(c);  }  } | | --- |  | class Demo {  public void m1(){  System.out.println("zero param");  }  public void m1(int a){  System.out.println("int param");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1();  d.m1(10);  }  } | class Demo {  public void m1(int b){  System.out.println("zero param");  }  public void m1(int a){  System.out.println("int param");  }  }  Here we will get errors because both methods have the same signature. | | --- | --- | | class Demo {  public void m1(){  System.out.println("zero param");  }  public void m1(float a){  System.out.println("int param");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1();  d.m1(10.2f);  d.m1(10);// statement 1  }  } | When we call a method first jvm will search for exact match i.e in this example it will search for m1(int), if jvm don’t get an exact match , then it will search for relative match.  Hence in this example at statement 1 , jvm will call m1(float) (relative match). | | class Demo {  public void m1(int b,float a){  System.out.println("first method");  }  public void m1(float a,int b){  System.out.println("second method");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1(10,10.5f);  d.m1(10.4f,20);  d.m1(10, 20);//error  }  } | class Demo {  public void m1(String s){  System.out.println("String method");  }  public void m1(Object a){  System.out.println("object method");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1(new Object());  d.m1("sam");  d.m1(null);//statement 1  }  }  At statement 1 , Jvm will call the String parameter method. Because null value can store into String and Object reference variable, in this situation jvm will give priority to child class i.e String class. | | class Demo {  public void m1(String s){  System.out.println("String method");  }  public void m1(StringBuffer a){  System.out.println("StringBuffer method");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1(new StringBuffer());  d.m1("sam");  d.m1(null);// Error  }  } | class Demo {  public void m1(String s){  System.out.println("String method");  }  public void m1(Object a){  System.out.println("Object method");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1(new Demo());  }  }  Here the Object param method will be called because Demo class object can store into Object class reference variable. | | class Animal {  }  class Monkey extends Animal{  }  class Demo {  public void m1(Animal s){  System.out.println("Animal method");  }  public void m1(Monkey a){  System.out.println("Monkey method");  }  public static void main(String[] args) {  Demo d = new Demo();  Animal a = new Monkey();  d.m1(a);  }  } | Here statement 1 , ‘a’ is a reference variable of Animal but it contains the object of Monkey.  So m1(Animal) method will call because in method overloading method resolution take place at  Compile time and depends on reference type. |   **Why is overloading called compile time polymorphism?**  Overloading is also called compile time polymorphism because in overloading takes place at compile time and depends on reference type.  Overloading is also called static polymorphism or early binding.  **Rules to overload a method:**   1. In overloading method return type can be anything, need not to be the same. 2. Method signatures should be different. 3. No restriction on static modifier. 4. No restriction on access specifier.   **Overriding:**  When child class is not satisfied with parent class implementation, child class will redefine that method in it’s own class.  Redefining parent class methods into child class is called overriding.  In overriding method signatures should be the same.   | class Test {  public void m1(){  System.out.println("parent method");  }  } | class Demo extends Test{  @Override  public void m1() {  System.out.println("child method");  }  } | | --- | --- | | class Test {  public void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  @Override  public void m1() {  System.out.println("child method");  }  public static void main(String[] args) {  Demo d = new Demo();  d.m1();//statement 1  Test t = new Test();  t.m1();//statement 2  Test t1 = new Demo();  t1.m1();// statement 3  }} | At statement 1, Demo class method will call.  In statement 2 , the Test class method will be called.  At statement 3 , Demo class method will call, because in overriding method resolution takes place at run time and depends on object type. |   **Why is method overriding also called run time polymorphism?**  In method overriding, method resolution takes place at run time and depends on object type, so it is called run time polymorphism.  Overriding is also called dynamic polymorphism or late binding.  Rules to override a method:   1. Return type should be the same or co variant. 2. Access specifiers should be the same or in increasing order. 3. Method signatures should be the same.   **Special cases:**  **Case 1:**  We cannot override static methods.   | class Test {  public static void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  public static void m1(){  System.out.println("child method");  }  } |  | | --- | --- |   Why can't we override static methods?  Static methods cannot be overridden because method overriding only occurs in the context of dynamic (i.e. runtime) lookup of methods. Static methods (by their name) are looked up statically (i.e. at compile-time).  **Case 2:**  **We cannot override private methods.**   | class Test {  private void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  private void m1(){  System.out.println("child method");  }  } | We cannot override private methods, because they are not visible to child class.  And overriding is a process where child class has a method from parent class and child class don’t want its implementation then child class will redefine that method.  In this example we will not get errors because here child class declared its own method m1(). | | --- | --- |   **Case 3:**  **If we want to stop overriding this method then declare the parent class method as final.**   | class Test {  public final void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  public void m1(){ System.out.println("child method");  }} | **Here** In child class we will get errors because Test class declared that method as final so we cannot modify that method in child class. | | --- | --- | | class Test {  public void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  public int m1(){  System.out.println("child method");  Return 10;  }  }  Here we have changed the return type, what will be the output? | class Test {  public Object m1(){  System.out.println("parent method");  return null;  }  }  class Demo extends Test{  public String m1(){  System.out.println("child method");  return null;  }  }  Here we have changed the return type, what will be the output? | | class Test {  public String m1(){  System.out.println("parent method");  return null;  }  }  class Demo extends Test{  public Object m1(){  System.out.println("child method");  return null;  }  }  Here we have changed the return type, what will be the output? | class Test {  public Test m1(){  System.out.println("parent method");  return null;  }  }  class Demo extends Test{  public Demo m1(){  System.out.println("child method");  return null;  }  }  Here we have changed the return type, what will be the output? | | class Test {  public void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  void m1(){  System.out.println("child method");  }  }  Here we have changed visibility then what will be the output? | class Test {  void m1(){  System.out.println("parent method");  }  }  class Demo extends Test{  public void m1(){  System.out.println("child method");  }  }  Here we have changed visibility then what will be the output? |   **Can we override the main method?**  Yes we can override the main method.   | class Test {  public static void main(String[] args) {  System.out.println("parent class method");  }  }  class Demo extends Test{  public static void main(String[] args) {  System.out.println("child class method");  }  } | Here if we run Test class then JVM will call Test class main method.  And if we run Demo class then JVM will run Demo class main method. | | --- | --- |   **Can we write a class without the main method?**  Yes. We can write a class without the main method but its parent class should have a main method otherwise we will get error **“Main method not found”.**   | class Test {  public static void main(String[] args) {  System.out.println("parent class method");  }  }  class Demo extends Test{    } | Output:  Parent class method  If we run Demo class then JVM will search for the main method in Demo class first, if it doesn't get, JVM will search it in parent class and will execute. | | --- | --- |   **Abstract Class:**  While declaring a method inside a class, if we don’t know what logic has to write inside that method then we will simply declare that method without logic.  If a method doesn't have logic then we have to declare that method as abstract.  Example:   | **Public abstract void m1();** | | --- |   If a class contains at least one abstract method then we have to declare that class as abstract.   | Public abstract class Demo{  Public abstract void m1();  Public void m2(){  // logic of m2 method here  }  } | | --- |   Abstract class is a collection of abstract methods and concrete methods.  Concrete method means method with some logic. In the above program m1() method is an abstract method and m2() is a concrete method.  **Who will provide logic for the abstract method** **of a class?**  To abstract method it’s child class will provide logic.   | Class Test{  Public abstract void m1();  } | Class Demo extends Test{  Public void m1(){  Sopln(“Hello”);  }  } | | --- | --- | | Class Test{  Public abstract void m1();  Public abstract void m2();  }  Class abstract Demo extends Test{  public void m1(){  System.out.println("Hello");  }  }  class Test1 extends Demo {  public void m2(){  System.out.println("Hello");  }  } | Here Demo class has implemented only m1() method so m2() method is still abstract in Demo, hence we have to declare Demo as abstract.  To m2() method Test1 which is the child of Demo will provide logic for m2() method.   | **Note:**  **In child class we have to give logic for all the abstract methods of parent class otherwise we have to declare that child class as abstract and it’s child class will give logic.** | | --- | |   Note: Even if a class does not contain an abstract method still we can declare that class as abstract.  We cannot instantiate an abstract class i.e we cannot create objects of an abstract class.  **Rules about abstract method:**   1. We cannot declare abstract methods as final.  | Public abstract class Demo {  public final abstract void m1();  }  Output: Error | For abstract methods we provide logic in child class i.e modify abstract method in child class.  But if we declare an abstract method as final, then final method we cannot modify. So child classes cannot give logic for abstract methods. | | --- | --- |  1. We cannot declare abstract methods as private.  | Public abstract class Demo {  public private abstract void m1();  }  Output: Error | For abstract methods we provide logic in child class but if we declare abstract method as private then this method will not be visible to child class then child class cannot give logic. | | --- | --- |  1. We cannot declare abstract methods as static.  | Public abstract class Demo {  public static abstract void m1();  }  Output: Error | Abstract method does not contain logic so if we call that method as className.method () then there is no use. | | --- | --- | |
| **Interface**  Interface is a pure abstract class in java.  Every method in interface is by default abstract. And every variable in interface is by default public static final.  Interface is a contract between client and service provider.  Example:   | Interface I {  Int a = 10;  Void m1();  } | | --- |   **Who will provide logic for interface methods?**  Its child classes will provide logic for interface abstract methods.   | Class Test implements I {  Void m1() {  System.out.println(“child implementation”);  }  Public static void main (String… args) {  Test t = new Test();  System.out.println (t.a);  t.m1 ();  }  } | | --- |  | Interface I1 {  Int a = 10;  }  Interface I2 {  Int a = 20;  } | Class Test implements I1, I2 {  Public static void main (String… args) {  Test t = new Test();  System.out.println ( t.a );//error ambiguity    // Alternative way  System.out.println (I1.a);  System.out.println (I2.a);  }  } | | --- | --- |  * We can implement more than one interface. * We cannot create an object of interface. * We cannot write constructor inside the interface. Because by default all the variables of interface are static. So we cannot define non-static variable inside interface, hence there is no need to write constructor. * Interface provides rules and guidelines to perform specific tasks.   **Marker Interface:**  An empty interface, if we implement that interface in your class your class object will get some extra ability, then that interface is called as Marker Interface.  Example:  1. Cloneable interface: If we implement a Cloneable interface in our class, our class object will get extra ability so that we can create clone of that object.  **2. RandomAccess:** If we implement the RandomAccess interface in Test class suppose, we can access data in Test class object fast and randomly.  **3. Serializable:**  If we implement a Serializable interface then we can serialize and deserialize our class object.  **Adapter class:**  **The** class which contains all the empty methods, is called the Adapter class.  Class Test {  Public void m1 () {  }  Public void m2 () {  }  } |
| **Java.lang package:**  This is the default package of java. If we use some classes from java.lang, we need not import the program because it is the default package in java.  This package contains Object class, String Class, StringBuffer class, StringBuilder class etc  **Object class:**  This is the default parent of every java class.  This class contains some methods which are required in every java class. That is why they have declared this class as default parent.  Object class has following methods:   1. toString() 2. hashcode() 3. equals() 4. finalize() 5. getClass() 6. clone() 7. wait() 8. wait(int milliseconds) 9. wait(int milliseconds) 10. notify() 11. notifyAll() 12. **toString():** This method returns String representation of object. When we print any reference variable, internally JVM calls the toString method.  | public class Demo1 {  int a = 10;  public static void main(String[] args) {  Demo1 d = new Demo1();  System.out.println(d);  }  } | Here , when print d variable, internally it will call tostring method as given below:  **System.out.println(d.toString())**  Whenever we print any reference variable, it will print **classname@hashcode in hexadecimal format**  It will not print data in that object. | | --- | --- | | public class Demo2 {  int a = 10;  public static void main(String[] args) {  Demo2 d = new Demo2();  System.out.println(d);  }  @Override  public String toString() {  return this.a+"";  }  } | If we want to print data from that object then we have to override the toString method as given in this example. | | public class Demo3 {  public static void main(String[] args) {  String s = new String("sam");  System.out.println(s);  }  } | Here, we are printing string reference variables. It should print **String@hashcode in hexadecimal format.**  But it will print data because String class has overridden toString method to display data. |   **2. hashCode() method**  Using this method we can get the hashcode of an object.  Hashcode is a unique value which we assign to an object.   | public class Demo1 {  public static void main(String[] args) {  Demo1 d = new Demo1();  System.out.println(d.hashCode());  }  } | Output:  1234567 that is some hashcode | | --- | --- | | public class Demo2 {  int a;  Demo2(int a){  this.a = a;  }  public static void main(String[] args) {  Demo2 d = new Demo2(101);  Demo2 d2 = new Demo2(102);  System.out.println(d.hashCode());  System.out.println(d2.hashCode());  }  @Override  public int hashCode() {    return this.a;  }  } | Here, we can override the hashcode method. |   **3. Equals method:**  The Equals method is used to compare references of objects.   | public class Demo3 {  int a = 10;  public static void main(String[] args) {  Demo3 d1 = new Demo3();  Demo3 d2 = new Demo3();  System.out.println(d1.equals(d2));  }  } | Here we will get output false.  Here, both the objects are having the same data still equals method will return false because equals method compares references not data. | | --- | --- | | public class Demo4 {  int a = 10;  public static void main(String[] args) {  Demo4 d1 = new Demo4();  Demo4 d2 = new Demo4();  System.out.println(d1.equals(d2));  }  @Override  public boolean equals(Object obj) {  Demo4 dx = (Demo4) obj;  if(this.a == dx.a){  return true;  }  return false;  }  } | I have override equals method to compare data of two objects. | | public class Demo5 {  public static void main(String[] args) {  String s1 = new String("sam");  String s2 = new String("sam");  System.out.println(s1.equals(s2));  }  } | Here, we are using equals method to compare two string variables so it will compare data not references because String class has overridden equals method to compare data. |   **Difference between == and equals method:**   1. **==**   **It is** an operator. If we use == operator to compare normal variables, it compares data.  If we use == operator to compare reference variables then it compares references.   | public class Demo1 {  public static void main(String[] args) {  int a = 10;  int b = 10;  System.out.println(a == b);  int c = 20;  int d = 30;  System.out.println(c == d);  }  }  Output:  True  false | public class Demo2 {  int a = 10;  public static void main(String[] args) {  Demo2 d1 = new Demo2();  Demo2 d2 = new Demo2();  System.out.println(d1 == d2);  }  }  Output:  False | | --- | --- |   **2. Equals method:**  It is a method present in Object class which is meant to compare references.  4. **Clone method:**  This method is used to create clones of an object. It does shallow cloning not deep cloning.   | public class Demo1 implements Cloneable{  int a = 10;  public static void main(String[] args) throws CloneNotSupportedException {  Demo1 d = new Demo1();  Demo1 d1 = (Demo1)d.clone();  System.out.println(d.a);  System.out.println(d1.a);  System.out.println(d.hashCode());  System.out.println(d1.hashCode());  }  } | class B{  int x = 10;  }  public class Demo2 implements Cloneable{  B b;  int y;  public Demo2(B b, int y) {  super();  this.b = b;  this.y = y;  }  public static void main(String[] args) throws CloneNotSupportedException {  B b = new B();  Demo2 d1 = new Demo2(b, 20);  Demo2 d2 = (Demo2)d1.clone();  System.out.println(d2.y);  System.out.println(d2.b.x);  }  }  Shallow cloning | | --- | --- | | class A{  int x = 10;  }  /\* This example explains deep cloning \*/  public class Demo3 {  A a;  int y;  public Demo3(A a, int y) {  this.a = a;  this.y = y;  }  public static void main(String[] args) throws CloneNotSupportedException {  A a = new A();  Demo3 d1 = new Demo3(a, 20);  Demo3 d2 = (Demo3)d1.clone();  d1.a.x = 150;  System.out.println(d1.a.x);  System.out.println(d2.y);  System.out.println(d2.a.x);  System.out.println(d1.hashCode());  System.out.println(d2.hashCode());  }  @Override  protected Object clone() throws CloneNotSupportedException {  A ax = new A();  ax.x = this.a.x;  Demo3 dx = new Demo3(ax, this.y);  return dx;  }  } | If we want deep cloning then we have to override the clone method. |   5. **getClass() method:**  This method will return the Class class object for that given object.  Class class object contains all class level information of a class.   | public class Demo1 {  int a = 10;  public static void main(String[] args) {  Demo1 d = new Demo1();  Class c = d.getClass();  System.out.println(c.getSuperclass());  }  //we can not override getClass method bcoz that is final  } | public class Demo2 {  public static void main(String[] args) {  String s = "sam";  Class c = s.getClass();  System.out.println(c.getSimpleName());  System.out.println(c.getSuperclass().getSimpleName());  }  } | | --- | --- | |
| **String class:**  This class we can use to store a group of characters.  We can create String class object using two ways:   1. Using String literal 2. Using new keyword  | public class Demo1 {  public static void main(String[] args) {  String s1 = "sam";  String s2 = new String("ram");  System.out.println(s1);  System.out.println(s2);  }  } | Output:  Sam  Ram | | --- | --- | | public class Demo2 {  public static void main(String[] args) {  String s1 = "sam";  String s2 = "ram";  String s3 = "sam";  System.out.println("Hashcode of s1 : "+s1.hashCode());//113631  System.out.println("Hashcode of s2 : "+s2.hashCode());//112670  System.out.println("Hashcode of s3 : "+s3.hashCode());//113631  System.out.println(s1 == s3);//true  System.out.println(s1.equals(s3));//true  }  } | Here, s1 and s3 will point to the same object because if we use String literal way to create String object, JVM creates object in String Constant Pool.  Whenever we create an object in a String constant pool, JVM will first check whether the pool is already having the object with the same data. If yes, it will assign that same object to that variable, it will not create a new object. But the object is not there in the pool then it will create a new object.  Hence, while creating string objects for data sam second time, it will not create new objects. It will assign s3 to old objects only. | | public class Demo3 {  public static void main(String[] args) {  String s1 = new String("sam");  String s2 = new String("sam");  System.out.println(s1.hashCode());//113631  System.out.println(s2.hashCode());//113631  System.out.println(s1.equals(s2));//true  }  } | If we use a new keyword to create string class objects, it will create new objects every time and that too in the heap area. And also it will create string objects in the String constant pool area. | | public class Demo4 {  public static void main(String[] args) {  String s1 = "sam";  System.out.println(s1.hashCode());//113631  s1 = "ram";  System.out.println(s1.hashCode());//112670    String s2 = "sam";  System.out.println(s2.hashCode());//113631  s2 = s2+" bye";  System.out.println(s2.hashCode());//1862450989  }  } | This example tells us, we cannot modify string object.  If we try to modify the String class object, it will create a new object.  That’s why string immutable. | | public class Demo5 {  public static void main(String[] args) {  String s1 = "sam";  System.out.println(s1.hashCode());//113631  s1 = "sam";  System.out.println(s1.hashCode());//113631  }  } | Output:  ?????????????? |   **StringBuffer class**  This class object is mutable, that is we can modify string buffer object.   | public class Demo1 {  public static void main(String[] args) {  StringBuffer sb = new StringBuffer("sam");  System.out.println(sb.hashCode());//366712642  sb.append(" ram");  System.out.println(sb.hashCode());//366712642  System.out.println(sb);//sam ram  }  } | public class Demo2 {  public static void main(String[] args) {  StringBuffer sb1 = new StringBuffer("sam");  StringBuffer sb2 = new StringBuffer("sam");  System.out.println(sb1.hashCode());//366712642  System.out.println(sb2.hashCode());//1829164700  }  } | | --- | --- | | public class Demo3 {  public static void main(String[] args) {  StringBuffer sb1 = new StringBuffer("sam");  System.out.println(sb1.hashCode());  sb1 = new StringBuffer("ram");  System.out.println(sb1.hashCode());  }  } |  | |
| **Wrapper classes:**  Every primitive data type has an internal wrapper class. Whenever we don’t want to use any data as value, if we want to use that data as object then we will use wrapper classes.  We can convert primitive value into wrapper object and vice versa.  There are two ways to convert normal value into wrapper object:   1. Using new keyword 2. Using valueOf method  | public class Demo1 {    public static void main(String[] args) {  Integer I1 = new Integer(10);  System.out.println(I1);//10  Integer I2 = new Integer("10");  System.out.println(I2);//10  Integer I3 = new Integer("ten");  System.out.println(I3);//NumberFormatException  }  } | Here, a new keyword will create an Integer class object and then in that object it will store that 10.  We can pass integer value, String value.  We can pass string value but that string should be number format string. Otherwise we will get NumberFormatException. | | --- | --- | | public class Demo2 {  public static void main(String[] args) {  Float f1 = new Float(10.5f);  System.out.println(f1);  Float f2 = new Float("10.5f");  System.out.println(f2);  Float f3 = new Float("10.5");  System.out.println(f3);  Float f4 = new Float(10.5);  System.out.println(f4);  }  } | public class Demo3 {  public static void main(String[] args) {  Character ch1 = new Character('a');  System.out.println(ch1);//a  //Character ch2 = new Character("a");//error  //Character ch3 = new Character(65);//error  }  } | | public class Demo4 {  public static void main(String[] args) {  Boolean b1 = new Boolean(true);  System.out.println(b1);//true  Boolean b2 = new Boolean(false);  System.out.println(b2);//false  Boolean b3 = new Boolean("true");  System.out.println(b3);//true  Boolean b4 = new Boolean("yes");  System.out.println(b4);//false  Boolean b5 = new Boolean("no");  System.out.println(b5);//false  }  } | If we pass any string other than true then it will convert that value into false. |   **valueOf method:**   | public class Demo1 {  public static void main(String[] args) {  Integer I1 = Integer.valueOf(10);  System.out.println(I1);//10  Integer I2 = Integer.valueOf("10");  System.out.println(I2);//10  Integer I3 = Integer.valueOf("1010");  System.out.println(I3);//1010  Integer I4 = Integer.valueOf("1010",2);  System.out.println(I4);//10  }  } | public class Demo2 {  public static void main(String[] args) {  Float f1 = Float.valueOf(10.5f);  System.out.println(f1);  Float f2 = Float.valueOf("10.5f");  System.out.println(f2);  Float f3 = Float.valueOf("10.5");  System.out.println(f3);  //Float f4 = Float.valueOf(10.5);//error  }  } | | --- | --- | | public class Demo3 {  public static void main(String[] args) {  Character ch1 = Character.valueOf('a');  System.out.println(ch1);  //Character ch2 = Character.valueOf(65);//error  //Character ch3 = Character.valueOf("a");//error  }  } | public class Demo4 {  public static void main(String[] args) {  Boolean b1 = Boolean.valueOf(true);  System.out.println(b1);//true  Boolean b2 = Boolean.valueOf("true");  System.out.println(b2);//true  Boolean b3 = Boolean.valueOf("yes");  System.out.println(b3);//false  Boolean b4 = Boolean.valueOf("no");  System.out.println(b4);//false  }  } |   **Wrapper Object to normal value:**  To convert a wrapper object into normal value we need to use xxxvalue method.   | public class Demo1 {  public static void main(String[] args) {  Integer I = new Integer(10);  int a = I.intValue();  System.out.println(a);  float f = I.floatValue();  long l = I.longValue();  short s = I.shortValue();  byte b = I.byteValue();  double d = I.doubleValue();  System.out.println(f+"..."+l+"..."+s+"..."+b+"..."+d);  }  } | public class Demo2 {  public static void main(String[] args) {  Float f = new Float(10.5);  float f1 = f.floatValue();  byte b = f.byteValue();  short s = f.shortValue();  int i = f.intValue();  long l = f.longValue();  double d = f.doubleValue();  System.out.println(f1+"---"+b+"---"+s+"---"+i+"---"+l+"---"+d);  }  } | | --- | --- | | public class Demo3 {  public static void main(String[] args) {  Double D = new Double(10.5);  byte b = D.byteValue();  short s = D.shortValue();  int i = D.intValue();  long l = D.longValue();  float f = D.floatValue();  double d = D.doubleValue();  System.out.println(b+"---"+s+"---"+i+"---"+l+"---"+f+"---"+d);  }  } | public class Demo4 {  public static void main(String[] args) {  Character ch = new Character('a');  char c = ch.charValue();  //we can not convert char into other type  }  } | | public class Demo5 {  public static void main(String[] args) {  Boolean b = new Boolean(true);  boolean b1 = b.booleanValue();  System.out.println(b1);  //we can not convert boolean into other type  }  } |  |   **Wrapper object to String conversion:**  To convert wrapper objects into string objects we have to use the toString method.   | public class Demo1 {  public static void main(String[] args) {  Integer I = new Integer(10);  String s1 = I.toString();  Float F = new Float(10.3);  String s2 = F.toString();  Character ch = new Character('a');  String s3 = ch.toString();  Boolean B = new Boolean(true);  String s4 = B.toString();  System.out.println(s1+"---"+s2+"---"+s3+"---"+s4);  }  } | | --- |   **String to normal value conversion:**  To convert string value to normal value we have to use parseXXX methods.   | public class Demo1 {  public static void main(String[] args) {  String s = "10";  int i = Integer.parseInt(s);  float f = Float.parseFloat(s);  byte b = Byte.parseByte(s);  short s1 = Short.parseShort(s);  double d = Double.parseDouble(s);  long l = Long.parseLong(s);  boolean b1 = Boolean.parseBoolean(s);  //char ch = Character.parseChar(s);//error  System.out.println(i+"---"+f+"---"+b+"---"+s1+"---"+d+"---"+l+"---"+b1);  }  } | | --- |   **Difference between new keyword and valueOf method:**    valueOf method, when it creates an object, first it will check whether that object is already in buffer or not. If the object is already in buffer, valueof method will not create a new object, it will assign the same object to the reference variable.  But in the case of a new keyword, every time a new keyword will create a new object.  Refer to the above diagram. |
| **JVM Architecture:**  **Virtual Machine**  --> Software simulation of a machine which can perform operations like a physical machine.  --> for example Calculator in our system  --> There are two categories of virtual machine:  a) HardWare or system based machine  b) Application or process based machine  --> system based machines provide several logical systems on the same computer with strong isolation from each other. that is on one physical machine we are defining multiple logical machines. The main advantage of hardware based VM is hardware resources sharing and improves utilizations of hardware resources.  Example KVM (Kernel Based Virtual Machine for linux system) , VMWare,cloud computing etc.  --> Application based machine acts as RunTime Engine to run a particular programming language application.  For example 1) JVM acts as a runtime engine to run java based applications.  2) PVM acts as runtime engine to run perl based applications  3) CLR acts as runtime engine to execute .net based applications  **JVM**  --> JVM stands for JAVA Virtual Machine and it is a part of JRE.  --> JVM acts as a runtime engine to run java based applications.  --> JVM performs two activities  a) Load .class file  b) Run .class file  -> Architecture diagram of JVM:    **ClassLoader SubSystem**  Class Loader subSystem is responsible for three activities:  a) Loading  b) Linking  c) Initialization  **a) Loading**  --> It means reading class files and stores corresponding binary data in the method area.  --> For each class file JVM will store corresponding information in the method area.  a) Fully qualified name of class  b) Fully qualified name of immediate parent class  c) Methods information  d) Variables information  e) Constructors information  f) Modifiers information  --> After loading immediately JVM will perform following task:  - In the heap area JVM will create Class class object to represent this total .class file information.  - Class class objects can be used by programmer to get class level information like methods info or variables info or constructor’s info etc.  - For example:  1. Class c = Class.forName ("Demo");  - Here forName () will load Demo.class file in method area and JVM will create Class class object which represents total info about Demo class and then that Class class object reference will store into c.  2. Demo d = new Demo ();  Class c = d.getClass ();  - Here getClass () will return reference of Class class object which represents total info about Demo.class file.  --> And only one Class class object will be created for any class.  --> For example  Demo d = new Demo ();  Class c = d.getClass ();  Demo d1 = new Demo ();  Class c1 = d1.getClass ();  Sopln (c.hashCode ()); //1010  Sopln (c1.hashCode ()); //1010 both the statement will display same reference  **b) Linking**  It consist of three activities:   1. Verify 2. Prepare 3. Resolution 4. **Verification or Verify:**   It is the process of ensuring that binary representation of a class is structurally correct or not. That is JVM will check whether the .class file generated by valid compiler or not i.e whether .class file is properly formatted or not.  When you are using C, C++ based softwares you will get some harmful application messages.  But in java you will not get such a message because Java is a secure language.  JVM contains a Bytecode Verifier which will check whether given .class file is generated by proper compiler or not. If that .class file is not generated by proper compiler then we will get java.lang.verifyError. JVM will not run that code.  Because of this java got more security and there is no chance of adding virus in java application.   1. **Preparation or prepare**   In this process JVM will allocate memory for class level static variables and assign default values.  Note: In the Initialization phase original values will be assigned to static variable. And here only default values will be assigned.   1. **Resolution**:   It is the process of replacing symbolic names in our program with original memory references from the method area.  Example  Class Test {  Main (String [] args) {  String s = new String (“sam”);  Student s = new Student ();  }  }  For above program following .class files will be loaded:   1. Object.class 2. String.class 3. Test.class 4. Student.class   The names of these classes are stored in the Constant pool of Test class. In the resolution phase these names are replaced with original memory level references from the method area.  **c) Initialization:**  In this all static variables are assigned with original values and static blocks will be executed from parent to child and from top to bottom.    **Types of Class Loader**  Class Loader Subsystem contains following three types of class loader:   1. Bootstrap class loader 2. Extension class loader 3. Application class loader or system class loader 4. **Bootstrap class loader:**   It is responsible to load core java API classes. That is the classes present in rt.jar.  It is responsible to load classes from bootstrap class path.  C:\Program Files\Java\jdk1.8.0\_91\jre\lib path represents bootstrap path.  It is by default available with every JVM and implemented in native languages like c or c++. And it is not implemented in java.   1. **Extension class loader**   It is responsible to load classes from the extension class path.  C:\Program Files\Java\jdk1.8.0\_91\jre\lib\ext represents extension class path.  It is implemented in java only. And corresponding class name is sun.misc.launcher$ExtClassLoader.   1. **Application class loader**   It is a child class of extension class loader. It loads class from application class path.  Application classpath means environment variable classpath.  It is developed in java only and it has a class name is sun.misc.Launcher$AppClassLoader.  **Working of class loader:**    When JVM wants to load .class file, JVM first will check whether requested .class file is already loaded in method area. If not, JVM will request a class loader subsystem. Then the Class loader subsystem will request the Application class loader. Application class loader will delegate request to extension class loader and extension loader will delegate to bootstrap class loader.  Bootstrap class loader will search requested .class file in bootstrap class path. If he doesn't get that .class file, he will delegate the request to the extension class loader.  Extension class loader will search that requested .class file in extension class path. If he doesn't get it, he will delegate the request to the Application class loader.  Application class loader search given .class file in application class path. If he finds it, he will load that .class file in the Method area and if he doesn't get that .class file then JVM will raise ClassNotFoundException.  Note: Class Loader follows delegation hierarchy principle.    **Memory Areas in JVM**     1. **Heap Area**   For every JVM one heap area is available. Heap area will be created at the time of JVM startup.  Objects and corresponding instance variables will be stored in the heap area. Every array in java is object only. Hence arrays also will be stored in the heap area.  Heap area can be accessed by multiple threads and hence data stored in the heap memory is not thread safe.  Heap area need not be continuous.  Program to display Heap Memory statistics:  A java application can communicate with JVM by using Runtime object.  Runtime class is present in java.lang package and it is a singleton class. We can create runtime object as follows:  Runtime runtime = Runtime.*getRuntime* ();  Once we got runtime object we can call the following methods on that object:  System.***out***.println (runtime.maxMemory ());  It returns the number of bytes of max memory allocated to the heap.  System.***out***.println (runtime.totalMemory ());  It returns total memory allocated to the heap (initial memory).  System.***out***.println (runtime.freeMemory ());  It returns a number of bytes of free memory present in the heap.   1. **Stack Area**   For every thread JVM will create a separate stack at the time of thread creation. Each and every method call performed by that thread will be stored in the stack including local variables also.  After completing a method, the corresponding entry from the stack will be removed. After completing all method calls the stack will become empty and that empty stack will be destroyed by the JVM just before terminating the thread.  Each entry in the stack is called stack frame or activation record.  The data stored in the stack is available only for the corresponding thread and not available to the remaining thread hence this data is thread safe.  **Stack Frame structure**  Each stack frame contains three parts:   1. Local Variable array 2. Operand stack 3. Frame data 4. **Local variable array:**   Public static void m1 (int a, int b){  Int m = 10;  }  Local variable array contains all parameters and local variables of the method. Each slot in the array is of four bytes. Values of type int, float and reference occupy one entry in the array.  Values of double \and long occupy two consecutive entries in the array. Byte, short and char values will be converted to int type before storing and occupy one slot. But the way of storing Boolean values varies from JVM to JVM. But most of the JVM follow one slot for Boolean values.   1. **Operand stack:**   Operand stack used by JVM as workspace. Some instructions can push the values to the operand stack and some instructions can pop values from operand stack and some instructions can perform required operations.   1. **Frame Data**   Frame data contains all symbolic references related to that method. It also contains reference to exception table which provides corresponding catch block information in the case of exceptions   1. **PC Register:**   For every thread a separate PC Register will be created at the time of thread creation. It contains the address of current executing instruction. Once instruction execution completes automatically PC register will be incremented to hold the address of next instruction.  This thing is internally used by JVM.   1. **Native Method Stack:**   For every thread JVM will create a separate native method stack. All native method calls invoked by that thread will be stored in the corresponding native method stack.  Native Method:  A Native method is a method which is implemented in a language other than java.  Steps to create native method:   1. Write java code 2. Compile java code 3. Create c header(.h file) 4. Write c code 5. Create shared code library 6. Run application   Example:  Public class Demo {  int i;  Public static void main (String[] args) {  Demo d = new Demo ();  d.i = 10;  d.test ();  }  //declare native method  Public native void test();  //load its library  Static {  System.*loadLibrary* ("LibraryName");  }  }   1. **Method area:**   Class loader subsystem loads .class file in Method area. Then JVM allocates static areas for given class in method area. All static variables will get memory in a static area.  Note:   1. Method area, heap area, stack area are considered IMP memory areas with respect to programmer. 2. Method area and Heap area are for JVM. For every JVM one per JVM. 3. Stack area , native method stack and PC Register are for thread. One per thread. 4. Instance variable will be stored in Heap area. 5. Static variables will be stored in the Method area. 6. Local Variables will be stored in the stack area.   **Execution Engine**  It is the central component of JVM. It is responsible to execute java class files. It mainly contain two components:   1. Interpreter 2. JIT Compiler 3. **Interpreter**   It is responsible to read byte code and interpret into machine code and execute that machine code line by line.  The problem with interpreters is that they interpret every time even the same method invoked multiple times which reduces the performance of the system. To overcome this problem, sun people introduced JIT compiler in 1.1 version.   1. **JIT Compiler**   Main purpose of JIT is to improve performance. Internally it maintains separate count for every method. Whenever JVM comes across a method call first method will be interpreted normally via interpreter and JIT increment corresponding count variable.  This process will continue for every method. Once the method count reaches threshold value then JIT identifies that method is repeatedly used. That method is called a hot spot.  Immediately JIT compiles that method and generates the corresponding native code. Next time when JVM comes across that method call then JVM uses native code directly and executes it instead of interpreting once again. So that performance of the system will be improved.  The threshold count varies from JVM to JVM.  Internally Profiler, which is the part of JIT compiler, is responsible for identifying hot spots.  Note: JVM interprets total program at least once. JIT compilation is applicable only for repeatedly required methods not for every method.  **Java Native Interface**  JNI acts as mediator for java method calls and corresponding native library. JNI is responsible for providing information about native libraries to JVM.  Native method libraries hold native library information. |
| **File Uploading**  Need of File:  Generally data stores in variable and objects presents only during execution. After execution all memory will get deallocated so we will lose data.  If we want to store data permanently we have to use permanent storage like file or database.  To work with file java has given IOStream api.  **Stream:**  Using stream we can read and write data into file.  Stream is a logical connection between java program and file.  Stream is a continuous data flow between java program and persistent storage.  **Types of streams:**   1. InputStream: It reads data from database to java program. 2. OutputStream: It sends data from java program to database.   **Types of java streams:**  In java we are allowed to send data through stream only either in byte or character format. Based on type of data passed through stream we have two types of stream:   1. Binary Stream: The stream which reads and writes data in the form of byte called Binary stream. 2. Character Stream: The stream which reads and writes data in the form of a character called a character stream.   To send and get data we have to create a stream between file and java program.  We can use several classes from java.io package to create stream i.e creating stream means creating object of one of that class.  Hence,  **Stream is a java object that allows reading and writing data to permanent storage.**  -> InputStream is a super class for all binary input stream classes.  -> OutputStream is a super class for all binary output stream classes.  -> Reader is a super class for all character input stream classes.  -> Writer is a super class for all character output stream classes.  **Note:** InputStream and Reader classes have a read () method to read data from file. Read () of InputStream reads byte by byte and read () of reader reads character by character.  **File class**   | **public** **class** Demo {    **public** **static** **void** main(String[] args){  File f = **new** File ("demo.txt");  System.***out***.println("Finish");  }  } | Here, if demo.txt file is already there then f will point to that file. Otherwise it will not create a new file. | | --- | --- | | **public** **class** Demo {    **public** **static** **void** main(String[] args) **throws** IOException{  File f = **new** File ("demo.txt");  **if** (!f.exists()) {  f.createNewFile();  }  System.***out***.println("Finish");  }  } | Here, exist () method returns true if the file is there. And it returns false if the file is not present.  CreateNewFile () method creates a new file with given name. |  | **Other methods from File class:**  **1.list () :** This method returns a list of names of all the files.  2. **isFile ():** This method returns true if f is pointing to file otherwise it will return false.  3. **isDirectory ():** This method returns true if f is pointing to folder otherwise it will return false. | | --- |   **FileWriter class:**   | **public** **static** **void** main(String[] args) **throws** IOException{  FileWriter fw = **new** FileWriter ("demo.txt");  System.***out***.println("Finish");  } | Here, FileWriter constructor will point to demo.txt file if file is there. If the file is not there, it will create a new file and will point to that file. | | --- | --- | | **public** **static** **void** main(String[] args) **throws** IOException{  FileWriter fw = **new** FileWriter ("demo.txt");  fw.write ("Sam");  fw.write ('a');  fw.write (99);  System.***out***.println("Finish");  } | Here, using FileWriter class we can write String value, character value only. If we write 99, it will write its ASCII character. | | **public** **static** **void** main(String[] args) **throws** IOException{  FileWriter fw = **new** FileWriter ("demo.txt",**true**);  fw.write("sam");  fw.write('a');  fw.write(99);  System.***out***.println("Finish");  } | Here, as we gave the second parameter true, it will append data. If it is false it will override previous data.  By default it is false. | | **public** **static** **void** main(String[] args) **throws** IOException{  File f = **new** File("demo.txt");  FileWriter fw = **new** FileWriter (f,**true**);  fw.write("sam");  fw.write('a');  fw.write(99);  System.***out***.println("Finish");  } | **public** **static** **void** main(String[] args) **throws** IOException{  File f = **new** File("demo.txt");  FileWriter fw = **new** FileWriter (f);  fw.write("sam");  fw.write('a');  fw.write(99);  System.***out***.println("Finish");  } |   **Drawbacks of FileWriter:**  **1.** We cannot write data other than String, character.  2. We cannot insert new line characters. Most of the system will not support new line characters.  BufferedWriter class:  Using this class, we can write new line characters. It is a child class of FileWriter.  We cannot pass the file name in the BufferedWriter class constructor. We can pass only FileWriter objects.   | **public** **static** **void** main(String[] args) **throws** IOException{  FileWriter fw = **new** FileWriter ("demo.txt");  BufferedWriter bw = **new** BufferedWriter(fw);  bw.write ("Sam");  bw.newLine ();  bw.write ('a');  bw.newLine ();  bw.write (99);  bw.close ();  } | Here, the newLine () method will add a new line in file.  Here, data will override with previous data. | | --- | --- | | **public** **static** **void** main(String[] args) **throws** IOException{  FileWriter fw = **new** FileWriter ("demo.txt", **true**);  BufferedWriter bw = **new** BufferedWriter(fw);  bw.write("Sam");  bw.newLine();  bw.write('a');  bw.newLine();  bw.write(99);  bw.close();  } | Here, data will append with previous data. |   **Drawbacks of BufferedWriter:**  **1.** We cannot write data other than string and character.  2. Writing a newLine method is complex.  **PrintWriter class:**  Using this class, we can write any type of data that is integer, float, double etc.  Adding a new line is not complex.   | **public** **static** **void** main(String[] args) **throws** IOException {  PrintWriter pw = **new** PrintWriter("demo.txt");  pw.write ("Sam");  pw.println ();  pw.println (10.5);  pw.println (10);  pw.println ('a');  pw.close();  } | | --- |   **FileReader class:**  Using this class, we can read data.   | **public** **static** **void** main(String[] args) **throws** IOException{  FileReader fr = **new** FileReader("demo.txt");  **int** a = 0;  **while** ((a= fr.read ())!=-1) {  System.***out***.println((**char**)a);  }  } | Read method will read a single character. And it will return its ASCII value and will move the cursor to the next character. When it reaches the end of file, it will return -1. | | --- | --- |   **Drawbacks of FileReader:**  We can read only character by character. We cannot read a single line at a time.  **BufferedWriter class:**  Using this class, we can read file data line by line.   | **public** **static** **void** main(String[] args) **throws** IOException{  FileReader fr = **new** FileReader("demo.txt");  BufferedReader br = **new** BufferedReader (fr);  String line = **null**;  **while** ((line = br.readLine ())!=**null**) {  System.***out***.println(line);  }  } | Here, the readLine method will read the total line and will move the cursor to the next line.  readLine method returns null, if it finds the end of file. | | --- | --- |   **Program to copy data from one file to another file:**   | **public** **static** **void** main(String[] args) **throws** IOException{  FileReader fr = **new** FileReader("demo.txt");  BufferedReader br = **new** BufferedReader(fr);  BufferedWriter bw = **new** BufferedWriter(**new** FileWriter("test.txt"));  String line = **null**;  **while** ((line = br.readLine())!=**null**) {  bw.write(line);  }  } | | --- |   **# FileInputStream and FileOutputStream**  These classes are used to read and write data as bytes from file.  Program:   | public class Test {  public static void main(String[] args) throws IOException {  FileOutputStream os = new FileOutputStream("test.txt");  os.write ('a');  os.write ('b');  os.write (99);  System.out.println("data successfully written in test.txt");  }  } | import java.io.\*;  public class Test {  public static void main(String[] args) throws IOException {  File f = new File("abc.txt");  // assume abc.txt file is already contains some data  FileInputStream is = new FileInputStream(f);  int i;  while((i = is.read())!= -1){  System.out.print((char)i);  }  }  } | | --- | --- |   Constructors:   1. FileInputStream(File file) 2. FileInputStream(String name)   **Program to copy file:**  **Copy data from test.txt into abc.txt**   | public class Test {  public static void main(String[] args) throws IOException {  FileInputStream is = new FileInputStream("test.txt");  FileOutputStream os = new FileOutputStream("abc.txt");  int i;  while((i = is.read())!= -1){  os.write(i);  }  System.out.println("data successfully written in test.txt");  }} | | --- |   Limitations:  These two allow us to read and write data only in the form of bytes. It is not possible to read or write data in the form of primitive type or objects.  **DataInputStream and DataOutputStream:**  These classes are used to read and write data as primitive types.  DataInputStream is for reading bytes from a binary input stream and convert them into corresponding primitive data type.  DataOutputStream is for converting any primitive type into a stream of bytes and writing these bytes to binary output stream.   | public class Test {    public static void main(String[] args) throws IOException {  FileOutputStream os = new FileOutputStream("demo.txt");  DataOutputStream out = new DataOutputStream(os);  out.writeInt(10);  out.writeFloat(10.5f);  out.write('a');  out.writeBytes("sam");  out.flush();  System.out.println("data has written to file");  }  } | public class Test {    public static void main(String[] args) throws IOException {  FileInputStream is = new FileInputStream("demo.txt");  DataInputStream out = new DataInputStream(is);  System.out.println(out.readInt());  System.out.println(out.readFloat());  System.out.println(out.readChar());  System.out.println(out.readByte());  System.out.println("data has written to file");  }  } | | --- | --- |   Note: readXXX() must call in the same order in which writeXXX() methods are called. Because readXXX() read a number of bytes from file based on data type.  **Limitations:**  Using these classes we can not write and read objects from file.  **ObjectOutputStream and ObjectInputStream:**  ObjectOutputStream is part of Java IO classes and its whole purpose is to provide us a way to convert java object to stream. When we create an instance of ObjectOutputStream, we have to provide the OutputStream to be used. This OutputStream is further used by ObjectOutputStream to channel the object stream to underlying output stream, for example FileOutputStream.  The object that we want to serialize should implement java.io.Serializable [interface](https://www.journaldev.com/1601/interface-in-java). Serializable is just a marker interface and doesn’t have any abstract method that we have to implement. We will get java.io.NotSerializableException if the class doesn’t implement a Serializable interface.   | **public** **class** Employee **implements** Serializable{  **private** **int** id;  **private** String name;  **private** **int** salary;  **public** Employee(**int** id, String name, **int** salary) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.salary = salary;  }  //getter and setter  } | | --- |  | **public** **class** WriterFile {  **public** **static** **void** main(String[] args) **throws** IOException {  Employee e = **new** Employee(101,"sam",1000);  FileOutputStream fos = **new** FileOutputStream("emp.ser");  ObjectOutputStream oos = **new** ObjectOutputStream (fos);  oos.writeObject (e);  System.***out***.println("done");  }  } | **public** **class** ReaderFile {  **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException {  FileInputStream fis = **new** FileInputStream("emp.ser");  ObjectInputStream ois = **new** ObjectInputStream(fis);  Employee e = (Employee) ois.readObject ();  System.***out***.println(e.getId ()+" "+e.getName ()+" "+e.getSalary ());  }  } | | --- | --- |   **Default Stream created in JVM**  In jvm by default three streams are created by class system to read data from keyboard and to write data to console. These streams are held by static final reference variable in system class.  These are:   1. In 2. Out 3. Err   System.out writes output to console and System.err writes exception to console.  System.in object connects to keyboard   | public class Test {    public static void main(String[] args) throws IOException {  System.out.println("enter one number");  int a = System.in.read();  System.out.println(a);  }  } | public class Test {    public static void main(String[] args) throws IOException {  System.out.println("Welcome");  System.err.println("Hello");  }  }  Here Welcome will print in black color and Hello will print in red color. | | --- | --- | |
| **Exception Handling:**  Exception is an unwanted situation because of which our program will terminate abnormally.  Exception is a predefined class.  Exception Hierarchy:  Image result for exception hierarchy in java  Exception and Error are both a predefined class.  Exception occurs due to mistake in logic.  For example:   | Class Test {  Main () {  System.out.println (“Hello”);  Int a = 10/0;  System.out.println (“finish”);  } | In the above program, before dividing he must have to check whether the second number is non zero or not. So here we will get an ArithmeticException exception. | | --- | --- |   Error occurs due to lack of system resources.  For example:   | Class Test {  Main () {  Test t = new Test ();  }  } | In this program, at runtime when JVM try to create an object of Test class, if JVM doesn't get enough memory in heap so it will through OutOfMemoryError.  So, errors occur due to lack of system resources. | | --- | --- |   There are two types of exceptions:  1. Checked Exception  2. Unchecked Exception   1. **Checked Exception:**   The exception which checks at compile time whether it is handled or not. It is called the Checked Exception.  For example: ClassNotFoundException   | Class Test {  Main () {  Class.forName (“Demo”);  }  } | Here, the forName method will load Demo.class file and JVM don’t get Demo.class at runtime, it will through ClassNotFoundException.  Exception will rise at runtime only but we have to handle it at compile time. Otherwise it will be through compile time error. | | --- | --- |  1. **Unchecked Exception:** The exception which will not check at compile time,   whether it is handled or not, is called an Unchecked exception.  For example:   | Class Test {  Main () {  Int a = 10/0;  }  } | Here, at runtime we will get ArithmeticException but it will not check at compile time whether it is handled or not. | | --- | --- |   **Note: Both Checked and unchecked exceptions will occur at runtime only.**  When an exception occurs, the program terminates abnormally. So to avoid this, we have to handle exceptions using try catch block.  Handling exceptions means not solving exceptions. It means providing an alternative way so that program can terminate normally.  The statements which can raise an exception, we will write into try block.  If an exception occurs in the try block then the catch block will get executed, otherwise catch block will not execute.   | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  System.***out***.println("Hello");  **try** {  **int** a = 10/0;  } **catch** (ArithmeticException e) {  System.***out***.println("catch block");  }  System.***out***.println("Finish");  }} | Here, in try block ArithmeticException will raise. That means ArithmeticException class object will be created. That object will pass to catch block. That's why we have written the ArithmeticException parameter in catch block. | | --- | --- |   **How to print information from an exception object?**   | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  System.***out***.println("Hello");  **try** {  **int** a = 10/0;  } **catch** (ArithmeticException e) {  e.printStackTrace ();//class name, description, location  System.***out***.println (e.toString ());//class name, description  System.***out***.println (e.getMessage ());//description  }  System.***out***.println("Finish");  }  } | | --- |   If in try multiple exceptions are going to occur then we can write multiple catch blocks. But at a time only one catch block can execute. Because if an exception occurs at the try block, remaining statements from the try block will not get executed.  For example:   | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  System.***out***.println ("Hello");  **int** b=0,c=0;  Scanner sc = **new** Scanner (System.***in***);  **try** {  System.***out***.println ("enter first number");  b = sc.nextInt ();  System.***out***.println ("enter second number");  c = sc.nextInt();  **int** a = b/c;//Arithmetic exception  String s = **null**;  System.***out***.println (s.length());//NullPointer Exception  } **catch** (ArithmeticException e) {  e.printStackTrace();  }**catch** (NullPointerException e) {  e.printStackTrace();  }  System.***out***.println("Finish");  }  } | | --- |  | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  System.***out***.println("Hello");  **int** b=0,c=0;  Scanner sc = **new** Scanner(System.***in***);  **try** {  System.***out***.println("enter first number");  b = sc.nextInt();  System.***out***.println("enter second number");  c = sc.nextInt();  **int** a = b/c;//Arithmetic exception  String s = **null**;  System.***out***.println(s.length());//NullPointer Exception  } **catch** (Exception e) {  e.printStackTrace();  }**catch** (NullPointerException e) {  e.printStackTrace();  }  System.***out***.println("Finish");  }  } | Here, we will get errors because the second catch block will never get executed.  That is an unreachable code.  **Note: we can write multiple catch blocks, but parent block should be last catch block. That is it should be in child to parent order.** | | --- | --- |   **Finally block:**  Finally the block executes whether an exception occurred or not. Every time finally the block will get executed.  Whatever resources we are creating inside try block, we can close them in finally block.  For example:   | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  System.***out***.println("Hello");  Connection con = **null**;  **try** {  con = DriverManager.*getConnection* ("");  } **catch** (SQLException e) {  e.printStackTrace ();  }**finally**{  **try** {  con.close ();  } **catch** (SQLException e) {  e.printStackTrace ();  }  }  System.***out***.println("Finish");  }  } | | --- |   **Throws keyword:**  Java **throws keyword** is used to declare an exception. It gives information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.  Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is the programmer's fault that he is not performing check-up before the code being used.   | **public** **class** Demo {  **public** **static** **void** m2() **throws** ArithmeticException{  System.***out***.println("m2 start");  **int** a = 10/0;  System.***out***.println("m2 end");  }  **public** **static** **void** m1() **throws** ArithmeticException{  System.***out***.println("m1 start");  *m2*();  System.***out***.println("m1 end");  }  **public** **static** **void** main (String[] args) {  System.***out***.println("main start");  **try**{  *m1*();  }**catch**(ArithmeticException e){  System.***out***.println("catch block");  }  System.***out***.println("main end");  }  } | Here, throws keyword will hand-over the exception object to the caller. That is from m2 method to its caller m1 method.  From m1 method to its caller main method.  Then the main method will handle exceptions.  Throws keyword will never handle exceptions, it will terminate program abnormally. So never use throws keywords in the main method.  It is used to convey the compiler that we have handled exceptions you can compile the program. | | --- | --- |   **Throw keyword:**  Throw keywords are used to generate exceptions manually.  For example:   | Class Test {  Main () {  System.out.println (“Hello”);  Throw new ArithmeticException ();  }  } | Here, throw keywords will generate ArithmeticException manually. | | --- | --- |   Most of the time, we use throw keyword to generate custom exception.  **Custom Exception:**  There is no predefined exception which can display user defined message. So, to display user defined message we can create custom exception class as given below:   | **public** **class** MyException **extends** RuntimeException {  **public** MyException(String s) {  **super**(s);  }  }  **public** **class** Demo {    **public** **static** **void** main (String[] args) {  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("enter first number");  **int** a = sc.nextInt();  System.***out***.println("enter second number");  **int** b = sc.nextInt();  **if** (b == 0) {  **try** {  **throw** **new** MyException("please don't enter second number as zero");  } **catch** (MyException e) {  System.***out***.println(e.getMessage());  }  }**else**{  **int** c = a/b;  System.***out***.println("division is "+c);  }  }  } | | --- |   **Try with resources:**  If we open a connection in try block then we have to close that connection in finally block.  But it is a headache to close the connection in finally block. Connection opened in the try block that should get close automatically. To do this, we can use try with resources as given below:   | **public** **class** Demo {    **public** **static** **void** main (String[] args) {  **try**(Connection conn = DriverManager.*getConnection*("")){  //use conn object here  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } | Here, we have used try with resources. Whatever resources created in the try block that will automatically close after try block execution.  This new feature came in 1.7 version. | | --- | --- |   **Difference between final, finally and finalize:**  **Final:** final is keyword if I use with variable we cannot modify its value, if I use with method we cannot override that method. If I use it with class we cannot extend that class.  **Finalize:** This method will be called by the garbage collector to perform a clean-up operation.  **Finally:** This is block. It is used to write a clean-up operation that is whatever resources created in try block that should be closed in finally block. |
| **Collection:**  **Need of collection:**  Suppose, we want to group multiple integer values. We can use array.  We can declare integer array to store multiple integer values as given below:  Int [] I = new int [5];  If we want to group multiple Employee object then we have to declare Employee array a given below:  Employee [] e = new Employee [5];  But in this Employee array, we cannot store other class objects until they are child of Employee class.  So, if we want to store any class object then declare Object array as given below:  Object [] o = new Object [5];  So, if we want to group multiple objects, we can use array but array has following drawbacks:  1. They are fixed in size.  2. We can store only homogeneous data.  3. Arrays don’t have inbuilt method support.  To overcome drawbacks of arrays we can use Collection.  **Need of Collection:**  It is used to group multiple objects into a single unit.  Collection is:  1. They are not fixed in size. They are dynamic in nature.  2. They can hold heterogeneous data.  3. They have inbuilt method support.  When we want to send multiple objects from one program to another program, we can send one by one but it is not good programming practice.  Here, we will collect all those objects into a Collection object and then we will send the collection object to another program.  H:\Notes\Diagram\Core Java\col_01_array_vs_col_01.png  H:\Notes\Diagram\Core Java\col_01_array_vs_col_02.png  **Collection is an interface.**  **Collection hierarchy:**  H:\Notes\Diagram\Core Java\col_02_hierarchy.png  **List interface:**  This interface we use when we want to group multiple objects into a single unit and we want to allow duplicates and want to preserve insertion order.  **ArrayList Class:**  This class we use when we want to group multiple objects into a single unit and we want to allow duplicates and want to preserve insertion order and we want to perform more read operations on that grouped data.   | **public** **static** **void** main(String[] args) {  ArrayList al = **new** ArrayList();  al.add(10);  al.add(30);  al.add(10);  al.add(10);  al.add(20.4);  al.add(10);  al.add(10);  al.add(20.4);  al.add(10);  al.add(10);  al.add(10);  System.***out***.println(al);  } | **Here,**  the new keyword will create an object of ArrayList class and it will allocate ten blocks initially. | | --- | --- |   ArrayList:  1.Default initial capacity: 10 blocks  2. Incremental Capacity: 3/2\*current capacity+1  3. Internal Data structure: Grow able Array  H:\Notes\Diagram\Core Java\col_03_arrayList.png   | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(30);  al.add(20);  ArrayList al1 = new ArrayList();  al1.add(100);  al1.add(300);  al1.add(200);  al1.addAll(al);  System.out.println(al1);  }  } | Here, using the addAll method we can add another arraylist object. | | --- | --- | | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(30);  al.add(20);  ArrayList al1 = new ArrayList(al);  System.out.println(al1); } | Here, we can pass another arraylist object while creating a new arraylist object as constructor parameter. | | public static void main(String[] args) {  ArrayList al = new ArrayList(15);  al.add(10);  al.add(30);  al.add(20);  System.out.println(al);  } | Here, if we use zero parameter ArrayList constructor, JVM will allocate 10 blocks initially.  If we want to allocate given number blocks initially then use parameterized constructor of ArrayList class. |   **Advantage of ArrayList class:**  It is best choice when we want to perform more read operations on stored data because ArrayList class internally implements RandomAccess marker interface.  **Drawbacks Of ArrayList class:**  **I**f we want to insert or delete data from the arraylist, more shifts will occur.  ArrayList is not recommended to use when we want to perform more update operations on stored data.  **LinkedList Class:**    We use LinkedList class when we want to allow duplicates, insertion order want to preserve and we want to perform more update operations on stored data.     | public static void main(String[] args) {  LinkedList ll = new LinkedList();  ll.add(10);  ll.add(20);  ll.add(30);  System.out.println(ll);  } | Here, when we add data, it will create a new node. | | --- | --- | | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(30);  al.add(20);    LinkedList ll = new LinkedList(al);  System.out.println(ll);  } | Here, If we want to add all the data from arraylist object to linkedlist object, we can pass arraylist object to linkedlist object. |   **Advantage of LinkedList class:**  This is the best option when we want to perform more update operations on stored data.  **Vector Class:**  Vector class we use when we want to allow duplicates, insertion order must be preserved and when we want to perform more read operation on stored data.     | public static void main(String[] args) {  Vector v = new Vector();  v.add(10);  v.addElement(20);  v.add(30);  v.add(15);  System.out.println(v.size());  System.out.println(v.capacity());  System.out.println(v);  } | Here, initially 10 blocks will get reserved.  Once 10 blocks get full, it will create new vector object with new size equal to 2 \* current capacity | | --- | --- | | public static void main(String[] args) {  Vector v = new Vector(25);  v.add(10);  v.addElement(20);  v.add(30);  v.add(15);  System.out.println(v.size());  System.out.println(v.capacity());  System.out.println(v);  } | Here, it will allocate 25 blocks initially. | | public static void main(String[] args) {  Vector v = new Vector(25,5);  v.add(10);  v.addElement(20);  v.add(30);  v.add(15);  System.out.println(v.size());  System.out.println(v.capacity());  System.out.println(v);  } | Here, it will allocate 25 blocks initially and incremental capacity will be 5. | | public static void main(String[] args) {    ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  Vector v = new Vector(al);  System.out.println(v);  } | If we want to add all the data from arraylist object to vector object then pass arraylist object to vector class constructor. |   **Advantage of Vector class:**   1. This is the best choice when we want to perform more read operations on stored data because internally Vector class implemented RandomAccess interface. 2. When we want synchronization then also we can use vector class because most of the methods of vector class are synchronized.   **Disadvantage of vector class:**  This is not recommended to use vectors when we want to perform more update operations on stored data.  **Difference between ArrayList and Vector:**   | **ArrayList** | **Vector** | | --- | --- | | 1. It is a non legacy class. | 1. It is a legacy class. The class which came with java 1.0 version is called legacy class. | | 2. Most of the methods are non synchronized. | 2. Most of the methods are synchronized. | | 3. Less security | 3. More security | | 4. High performance | 4. Low performance | | 5. Incremental capacity = 3/2\*current capacity + 1 | 5. Incremental capacity = 2 \* current capacity |   **Stack Class:**  This class we will use when we want to operate data in first-in-last-out order.     | public static void main(String[] args) {  Stack s = new Stack();  s.push(10);  s.push(20);  s.push(30);  s.add(40);  Stack s1 = (Stack)s.clone();  System.out.println(s1);  System.out.println(s);  System.out.println(s.search(10));  System.out.println(s.pop());  System.out.println(s.peek());  System.out.println(s.capacity());  } | Here,  Push method will add data into stack.  Pop method will read and remove the top element from the stack.  The Peek method will read the top element from the stack.  Search method will return the offset of the given data.  Offset is the position of that element from top. | | --- | --- |   **Set interface:**  We use this interface when we don’t want duplicates and we don’t bother about insertion order.  **HashSet class:**  We use this class when we want to store unique elements and don’t want to allow insertion order.    In HashSet:   1. Duplicates will not be allowed. 2. Insertion order will not preserve because elements will insert according to its hashcode. 3. If we try to add duplicates, add method will return false and will not add that duplicate data. But we will not get any error.  | public static void main(String[] args) {  HashSet h = new HashSet();  h.add(10);  h.add(20);  h.add(30);  h.add(40);  h.add(10);  System.out.println(h);  } | Here, an internally hash table of 16 buckets will create a predefined set of hashcodes.  Default initial capacity is 16 and default load factor is 0.75%.  Elements will insert according to its hashcode, so insertion order will not preserve.  When we try to add 10 for the second time, it will not add. Add method will return false. | | --- | --- | | public static void main(String[] args) {  HashSet h = new HashSet(20);  h.add(10);  h.add(20);  h.add(30);  System.out.println(h);  } | Here, initially a hash table with 20 buckets will be created. | | public static void main(String[] args) {  HashSet h = new HashSet(20, 0.90f);  h.add(10);  h.add(20);  h.add(30);  System.out.println(h);  } | Here, the hash table with 20 buckets will be created initially. When 90 % of the hash table gets full, a new hash table will be created. | | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  al.add(10);  HashSet h = new HashSet(al);  System.out.println(h);  } | Here, if we want to add all the data from arraylist to hashset, we can pass arraylist object to HashSet constructor.  While adding data from arraylist to hashset, add method will remove all the duplicate data. |   **Disadvantage of HashSet class:**  It does not preserve insertion order.  **LinkedHashSet class:**  This class does not allow duplicates and insertion order will be preserved.   | public static void main(String[] args) {  LinkedHashSet h = new LinkedHashSet();  h.add(10);  h.add(20);  h.add(30);  System.out.println(h);  } | public static void main(String[] args) {  LinkedHashSet h = new LinkedHashSet(20);  h.add(10);  h.add(20);  h.add(30);  System.out.println(h);  } | | --- | --- | | public static void main(String[] args) {  LinkedHashSet h = new LinkedHashSet(20,0.90f);  h.add(10);  h.add(20);  h.add(30);  System.out.println(h);  } | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  al.add(10);  LinkedHashSet h = new LinkedHashSet(al);    System.out.println(h);  } |   **TreeSet class:**  When we don’t want duplicates and we want to insert elements according to sorting order of elements then use TreeSet class.     | public static void main(String[] args) {  TreeSet t = new TreeSet();  t.add(10);  t.add(5);  t.add(13);  t.add(7);  t.add(11);  t.add(4);  t.add(15);  t.add(5);  System.out.println(t);  } | Here, elements will be stored according to natural sorting order.  Here add method will call compareTo method of Comparable interface to compare data.  In TreeSet we cannot add heterogeneous data because we need to compare data. | | --- | --- | | public class MyComparator implements Comparator{  @Override  public int compare(Object o1, Object o2) {  Integer I1 = (Integer) o1;  Integer I2 = (Integer) o2;  if (I1<I2) {  return 1;  }else if(I1>I2){  return -1;  }else{  return 0;  }  }  }  ----------------------------------------------  public static void main(String[] args) {  TreeSet t = new TreeSet(new MyComparator());  t.add(10);  t.add(5);  t.add(13);  t.add(7);  t.add(11);  t.add(4);  t.add(15);  t.add(5);  System.out.println(t);  } | Here, in TreeSet constructor we have passed the comparator object, so now add method will not call compareTo method for comparison.  It will compare methods of MyComparator class.  If we want a custom sorting order then pass the comparator object to TreeSet constructor and write comparison logic in compare method. | | public class Product implements Comparator<Product>{  int price;  String name;  public Product(int price, String name) {  super();  this.price = price;  this.name = name;  }  public Product() {  // TODO Auto-generated constructor stub  }  @Override  public int compare(Product p1, Product p2) {  if(p1.price>p2.price){  return -1;  }else if(p1.price<p2.price){  return 1;  }else{  return 0;  }  }    } | public class Demo3 {  public static void main(String[] args) {  Product p1 = new Product(100, "prod1");  Product p2 = new Product(50,"prod2");  Product p3 = new Product(150,"prod3");  List<Product> al = new ArrayList<Product>();  al.add(p1);  al.add(p2);  al.add(p3);  Collections.sort(al,new Product());  for(Product p :al){  System.out.println(p.name);  }  }  } |   **Cursors In Collection:**  We can use cursor to read data from the Collection.  There are three types of cursors in Collection:   1. Enumeration 2. Iterator 3. ListIterator 4. **Enumeration**  | public static void main(String[] args) {  Vector v = new Vector();  v.add(10);  v.addElement(20);  v.add(30);  v.add(15);  Enumeration e = v.elements();  while (e.hasMoreElements()) {     |  |  | | --- | --- |   int a = (int) e.nextElement();  System.out.println(a);  }  } | hasMoreElements method returns moves cursor to the next record and return true if record is available there. If a record is not available there it will return false.  the nextElement method will return data from where cursor is pointing.  Return type of this method is Object so we have to type cast it. | | --- | --- | --- | --- |   **Drawbacks of Enumeration:**   1. We can use this cursor only with legacy classes. 2. We can only read the data. We cannot add or remove data from collection using this cursor.   **2. Iterator:**  This is a universal cursor. We can apply for legacy and non legacy classes.   | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  Iterator i = al.iterator();  while (i.hasNext()) {  int a = (int) i.next();  if(a == 20){  i.remove();  }  }  } | hasNext method returns moves cursor to the next record and returns true if record is available there. If a record is not available there it will return false.  the next method will return data from where the cursor is pointing.  Return type of this method is Object so we have to type cast it. | | --- | --- |   Using Iterator we can read and remove data.  **Drawbacks of Iterator:**   1. Using Iterator we cannot add data. 2. Using an Iterator we cannot read in a backward direction. It is a one dimensional cursor.   **3. ListIterator:**  This is a bidirectional cursor. In both directions we can read data.   | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  ListIterator i = al.listIterator();  while (i.hasNext()) {  int a = (int) i.next();  if (a == 20) {  i.add(25);  }  if( a == 30){  i.remove();  }  }  } | hasNext method moves cursor to the next record and returns true if record is available there. If a record is not available there it will return false.  the next method will return data from where the cursor is pointing.  hasPrevious method moves cursor to previous record and returns true if record is available there. If a record is not available there it will return false.  the previous method returns a record where cursor is pointing. | | --- | --- |   **Different ways to read collection data:**  **Using cursors**  **Using get method**   | public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add(10);  al.add(20);  al.add(30);  for (int i = 0; i < al.size(); i++) {  System.out.println(al.get(i));  }  } | | --- | | **4. Using enhanced for loop:**  public static void main(String[] args) {  ArrayList<Integer> al = new ArrayList<Integer>();  al.add(10);  al.add(20);  al.add(30);  for(Integer i:al){  System.out.println(i);  }  } |   **Generics**  Main objective of generics is to provide type safety and to resolve type casting problems.  **Type safety:**   | class Test{  public static void main(String[] args) {  String[] s = new String[5];  s[0] = "sam";  s[1] = new Integer(10);  }  } | we will get errors because we cannot store integer objects into String array.  Here we will get a compile time error.  Arrays are type safe. I.e we can give the guarantee for the type of elements present inside the array.  For example if our programming requirement is to hold only String type of objects , we can choose String array. BY mistake if we are trying to add any other type of object we will get compile time error.  But collections are not type safe. We can not give the guarantee for the type of element present inside the collection. For example if our programming requirement is to hold only string type of objects and if we choose arraylist , by mistake if we are trying to add any other type of object we will not get any compile time error but the program may fail at runtime. | | --- | --- | | class Test{  public static void main(String[] args) {  ArrayList al = new ArrayList();  al.add("sam");  al.add(10);  String name = (String)al.get(0);  String name2 = (String)al.get(1);//statement1  System.out.println();  }  } | Here we will get runtime exception at statement 1.  Due to this collections are not safe to use with respect to type.i.e collections are not a type safe.  Hence we can say arrays are type safe so when we want to have type safety then we can use array then what is the need of generics?  Arrays are having some problems so when we want to have type safety then we should go for generics.  Main purpose of Generics is to provide type safety and to resolve type casting problems. |   Generics:  By default Collection is not type safe but we can achieve type safety as given below:   | public class Demo {  public static void main(String[] args) {  ArrayList<String> al = new ArrayList<String>();  al.add ("sam");  al.add(10);//statement 1  }  } | Here we can add only String objects in arraylist al. At statement 1 we will get compile time error.  Hence though generics we are getting type safety. | | --- | --- |   Generics resolves type casting problems also as given below:   | public class Demo {  public static void main(String[] args) {  ArrayList<String> al = new ArrayList<String>();  al.add("sam");  al.add("ram");  String name = al.get(0);  }  } | Here at the time of retrieval we are not required to perform type casting.  Hence through generics we can solve type casting problems. | | --- | --- |  | 1. ArrayList<String> al = new ArrayList<String>(); | 2. List<String> al = new ArrayList<String>(); | | --- | --- | | 3. Collection<String> al = new ArrayList<String>(); | 4. ArrayList<Object> al = new ArrayList<String>(); |   Here at example 2 and 3 we will not get any error because List and Collection are the parent class of ArrayList and parent class reference can hold child class object.  But in example 4 we will get errors because in the case of parameter we can not use parent child relation. Parameters should be the same.  **Map Interface:**  When we want to group multiple objects in the form of key value, we use Map interface.  In map,   1. Data will be store as key value 2. Key and value both will be stored as objects. 3. Values can be duplicate 4. Keys can not be duplicate 5. Keys and values can be heterogeneous. 6. We can add any number of null values. 7. We can add only one null key.   **HashMap Class:**   | public static void main(String[] args) {  HashMap h = new HashMap();  h.put("name", "sam");  h.put("rollno", 101);  h.put("marks", 75.20);  h.put("age", null);  h.put("null", null);  System.out.println(h);  } | Here, the hash table will be created with initial capacity 10. | | --- | --- | | public static void main(String[] args) {  HashMap h = new HashMap(20);  h.put("name", "sam");  h.put("rollno", 101);  h.put("marks", 75.20);  System.out.println(h);  } | Here, the hash table will create with initial capacity 20. |   In the hash table, insertion order will not preserve because elements will insert according to hashcode of keys.  **LinkedHashMap class:**  In this class, elements will insert according to insertion order.   | public static void main(String[] args) {  LinkedHashMap<String, String> h = new LinkedHashMap<String, String>();  h.put("name", "sam");  h.put("rollno", "101");  h.put("age", "12");  System.out.println(h);  } | | --- |   **TreeMap Class :**  This class stores data according to sorting order of keys.   | public static void main(String[] args) {  TreeMap<String, String> h = new TreeMap<String, String>();  h.put("name", "sam");  h.put("rollno", "101");  h.put("age", "12");  System.out.println(h);  } | It will insert data according to natural sorting order.  If we want a custom order, then use a comparator. | | --- | --- | |
| **MultiThreading**  **MultiTasking:** Executing multiple tasks simultaneously is called Multitasking.  Multitasking has two types:   1. Process based multitasking: Executing multiple tasks simultaneously where each task is separate independent process is called Process based multitasking. This is an operating system related. 2. Thread based multitasking: Executing multiple tasks simultaneously where each task is separate independent part of same program. This is threading related concept.   Consider, we have a program with 10 thousands of line of code. In that, first 5 thousand lines and second 5 thousand lines of code are independent. There is no relation between them.  In this case, we can create two threads one thread will execute the first 5 thousands lines of code and the second thread will execute the second 5 thousand line of code. And both threads will execute simultaneously.  Executing multiple threads simultaneously is called multi threading.  What is thread?  It is a lightweight process that executes some tasks.  Or  It is a simple java class which extends Thread class.  For example:   | public class MyThread extends Thread{  @Override  public void run() {  for (int i = 0; i < 100; i++) {  System.out.println("Hello");  }  }  } | Here, MyThread is a new Thread. We have to override the run method.  Run method contains the job of the thread.  The job of my thread is to display Hello 10 times.  But this is just imagination, if we want to execute that task, we have to start the thread. | | --- | --- | | public class MyThread extends Thread{  @Override  public void run() {  for (int i = 0; i < 100; i++) {  System.out.println("Hello");  }  }  } | public class Test {  public static void main(String[] args) {  MyThread t = new MyThread();  t.start();  for (int i = 0; i < 100; i++) {  System.out.println("main method");  }  }  } |   **Explanation**:  When we run the Test class, the main thread which is present in JVM will call the main method. Main thread will execute all the code present in the main method.  When we call the t.start method, Thread class start method will execute.  Thread class start method will do following three task:   1. Create thread MyThread. 2. Register that thread with thread scheduler. 3. Call run method of MyThread whenever thread scheduler allocates processor to MyThread.   After t.start (), Thread Scheduler will have two threads, main thread and mythread.  Now depending on how Thread Scheduler allocates processor to threads, we will get output.  Here we will get mixed output as given below:   | Main method  Main method  Hello  Main method  Hello  Hello  .  .  . | | --- |   In multi threading, we cannot predict output because all threads will execute simultaneously.  **How to get the name of a thread executing a particular code?**   | public class Test {  public static void main(String[] args) {  MyThread t = new MyThread();  t.start();  System.out.println("main thread name: "+Thread.currentThread().getName());  System.out.println("new thread name: "+t.currentThread().getName());  }  } | the currentThread method will return the object of current thread which is executing that particular line.  getName method will return the name of that thread.  Output:  Main thread name: main  New thread name: Thread-0 | | --- | --- |   **Thread Priority:**  Every thread has some priority. While allocating processors, Thread Scheduler will check priority of those threads. Thread with high priority will have more chances to get a processor.  In the above program, there are two threads: main and t. Main thread is the parent thread and t is the child thread.  By default, the main thread has priority 5. Same priority will inherit to t thread.  **How to get thread priority?**   | public class Test {  public static void main(String[] args) {  MyThread t = new MyThread();  t.start();  System.out.println("main thread priority: "+Thread.currentThread().getPriority());  System.out.println("new thread priority: "+t.getPriority());  }  } | Output:  Main thread priority: 5  New thread priority: 5  getPriority method will return priority of that thread. | | --- | --- |   **How to set priority to thread?**   | public class Test {  public static void main(String[] args) {  MyThread t = new MyThread();  t.start();  Thread.currentThread().setPriority(7);  t.setPriority(9);  System.out.println("main thread priority: "+Thread.currentThread().getPriority());  System.out.println("new thread priority: "+t.getPriority());  }  } | Output:  Main thread priority: 7  New thread priority: 9  setPriority method will set the priority to the thread.  Here, the new thread will get more processor. | | --- | --- |   **Thread join () method:**  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(); it causes the current thread to pause its execution until thread it joins completes its execution.  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 MyThread extends Thread{  static int total = 0;  @Override  public void run() {  for (int i = 0; i < 100; i++) {  total = total + i;  }  }  }  ----------------------------------------------  public class Test {  public static void main(String[] args) throws InterruptedException {  MyThread t = new MyThread();  t.start();  t.join();  System.out.println(MyThread.total);  }  } | Here, the main thread should wait till the execution of t thread because t thread add 0 to 100 numbers.  Till this addition get complete main thread should wait until t thread completes its execution.  So we will call the t.join () method before accessing the total in the main thread.  Here, the main thread will till complete execution of t.  If we don’t want to wait till the complete compilation of t thread then specify specific time as given below:  T.join (10000) -> in this case the main thread will wait for 10 sec then it will start its execution.  We can call t.join (10000,100) ->  Now the main thread will wait 1000 millisecond and 100 nano second. | | --- | --- |   **Sleep method in Thread class:**  Thread.sleep() method can be used to pause the execution of current thread for specified time in milliseconds.   | public class Test {  public static void main(String[] args) throws InterruptedException {  for (int i = 0; i < 10; i++) { Thread.sleep(1000);  System.out.println("main thread");    }  }  } | Here the main thread will sleep for 1 sec every time before printing the main thread. | | --- | --- |   **Synchronized block**  When very few lines of the code require synchronization then it’s not recommended to declare the entire method as synchronized we have to write that logic inside synchronized block.  The main advantage of synchronized block over synchronized method is it reduces waiting time of threads and improves performance of the system.   | synchronized (this) {    }  If a thread got a lock of current object then only it is allowed to execute this area. | | --- | | synchronized (b) {    }  If a thread got lock of a particular object b then only it is allowed to execute this area. | | synchronized (Display.class) {    }  If a thread got class level lock of Display class then only it is allowed to execute this area code. | | Int x = 10;  synchronized (x) {    }  Here we will get errors because we can not pass primitive values to synchronized blocks. |   **Race Condition**  If multiple threads are operating simultaneously on the same java object then there may be a chance of data inconsistency problem.  This is called Race condition.  We can overcome this problem by using synchronized keywords.  **What is object lock and when it is required?**  Whenever thread wants to execute a non static synchronized method, the that thread needs object lock.  **What is class lock and when it is required?**  Whenever thread wants to execute a static synchronized method then the thread requires class lock.  **Inter Thread Communication**  Two threads can communicate with each other by using wait(), notify() and notifyAll() methods.  The thread which is expecting an update is responsible to call the wait method then immediately the thread will enter into waiting state.  The thread which is responsible to perform the update ,after performing the update it is responsible to call the notify method then the waiting thread will get that notification and continue its execution with those updated items.  Why wait , notify, notifyAll() methods are available in Object class ?  Because thread can call these methods on any java object.  To call wait or notify or notifyAll methods on any object, thread should be the owner of that object i.e the thread should have lock of that object i.e the thread should be inside synchronized area.  Hence we can call these methods only from synchronized areas otherwise we will get runtime exception saying IllegalMonitorStateException.  If a thread calls the wait method on any object, it immediately releases the lock of that particular object and enters into waiting state.  If a thread calls notify() method on any object it releases the lock of that object but may not immediately. Except wait notify and notifyAll no other method where thread release the lock.   | package com.nt.demo;  class MyThread extends Thread{  int total = 0;  @Override  public void run() {  synchronized (this) {  System.out.println("new thread started calculation");  for (int i = 0; i < 100; i++) {  total = total + i;  }  System.out.println("new thread done with calculation");  this.notify();  **}**  **}**    **}** | package com.nt.demo;  class Demo{  public static void main(String[] args) throws InterruptedException{  MyThread t = new MyThread();  t.start();  synchronized (t) {  System.out.println("main thread called wait method");  t.wait();  System.out.println("main thread got notification");  System.out.println("total "+t.total);  }  }  } | | --- | --- |   If we call Thread.sleep(10000) then at the time of sleeping if that thread gets notification then sleeping thread won’t receive that notification so waiting thread will go into waiting forever.  **Green Thread**  Java multithreading concept is implemented by using the following two models:   1. Green thread model 2. Native os model 3. **Green Thread model:** The thread which is managed completely by JVM without taking support of OS, called Green Thread. Few OS like Solaris provides support for green thread models. This model is deprecated not recommended to use. 4. **Native OS Model:** The thread which is managed by the JVM and OS both is called Native OS modal. All windows based os provide support for this modal.   **Thread Pools(Executor Framework)**  Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.  Since active threads consume system resources, a [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.  A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing. Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive   * Java provides the Executor framework which is centered around the Executor interface, its sub-interface –ExecutorService and the class-ThreadPoolExecutor, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute. * They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics. * To use thread pools, we first create an object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows you to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.      | newFixedThreadPool(int) | Creates a fixed size thread pool. | | --- | --- | | newCachedThreadPool() | Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available | | newSingleThreadExecutor() | Creates a single thread. |   In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.  We can shutdown executorservice by using shutdown method as given below:  ser.shutdown();   | package com.nt.demo;  class MyThread implements Runnable{  String name;  public MyThread(String name) {  this.name = name;  }  @Override  public void run() {  System.out.println("Hello "+name);  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  System.out.println("Bye "+name);  }    } | package com.nt.demo;  import java.util.concurrent.Executor;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  class Demo{  public static void main(String[] args) throws InterruptedException{    ExecutorService service = Executors.newFixedThreadPool(3);  for (MyThread myThread : m) {  service.submit(myThread);  }  service.shutdown();  }  } | | --- | --- |  |  | **After executing first three tasks** | | --- | --- |   Risks in using Thread Pools   * + 1. [Deadlock](https://www.geeksforgeeks.org/deadlock-in-java-multithreading/) : While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.     2. Thread Leakage :Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.     3. Resource Thrashing :If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.   Important Points   * + 1. Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.     2. Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.     3. The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.     4. One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.  **Starvation**  If a thread is not granted CPU time because other threads grab it all, it is called "starvation". The thread is "starved to death" because other threads are allowed the CPU time instead of it. The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute. **Causes of Starvation in Java** The following three common causes can lead to starvation of threads in Java:   1. Threads with high priority swallow all CPU time from threads with lower priority. 2. Threads are blocked indefinitely waiting to enter a synchronized block, because other threads are constantly allowed access before it. 3. Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it. |
| **More on String class:**  **String** is a predefined class in java.  Whenever we want to represent a group of characters as objects then we can use String class.   | public class Test {    public static void main(String[] args) {  String s = "sam";  s.concat(" abc");  System.out.println(s);  }  }  Output: sam | In the above program, we are trying to modify the String object. When we try to modify string objects, it will create new objects. We can not modify the same object.  That’s why string is immutable. | | --- | --- |   Class is immutable, when we create an object once, we should not be able to modify it.   | public class Test {    public static void main(String[] args) {  String s = "sam";  s = s.concat(" abc");  System.out.println(s);  }  }  Output: sam abc |  | | --- | --- |   **Different ways to create String object:**  There are two ways using which we can create string object given below:   1. Using string literal 2. Using new keyword   **String literal way:**   | public class Test {    public static void main(String[] args) {  String s1 = "sam";  String s2 = "ram";  String s3 = "sam";  System.out.println(s1.hashCode());  System.out.println(s3.hashCode());  System.out.println(s1 == s3);  }  }  Output:  1010  1010  true |  | | --- | --- |   **In the above program,** s1 and s3 are having the same hashcode because when we use String literal way to create string object, it creates object of string in string constant pool.  When Jvm creates an object in a pool, first it will check whether the pool is already having an object with the same data or not. If not, it will create a new object. If the object is already there, in that case it will not create a new object. That’s why in the case of s3, jvm has not created an object.  Note: When we use string literal way to create object of string, jvm will create object in string constant pool.  String constant pools can not have duplicate objects.  **Creating string object using new keyword:**   | public class Test {    public static void main(String[] args) {  String s1 = new String("sam");  String s2 = new String("sam");  String s3 = new String("ram");  System.out.println(s1 == s2);  }  } |  | | --- | --- |   When we use a new keyword to create an object, jvm will create a String object in heap area. Actually, it will create two objects, one in the string pool area and one in the heap area as given in the above diagram. But the variable will point to the object which is created in the heap area.  New keyword will create one extra object in the pool for future use.  Here, s1 and s3 will have the same hashcode but different reference. Because in the String case, hashcode will be generated by manipulating characters in that string. SO s1 and s3 variables have the same string “sam”, so in this case hashcode will be same but reference will be different.   | public class Test {  public static void main(String[] args) {  String s1 = new String("sam");  String s2 = new String("sam");  System.out.println(s1 == s2);  System.out.println(s1.equals(s2));  }  }  Output: false  true |  | | --- | --- |   In the above program, as I am using a new keyword, jvm will create two objects in the heap area and one in the string pool area.  S1 == s2 will return false because == compares references and s1 and s2 are pointing to two different objects, so reference will be different. So == operator will return false.  Note: here s1 and s2 will have the same hashcode.  s1.equals(s2) will return true. Because equals method is overridden in String class for data comparison. So as s1 and s2 contain the same data, it will return true.   | public class Test {  public static void main(String[] args) {  StringBuffer s1 = new StringBuffer("sam");  StringBuffer s2 = new StringBuffer("sam");  System.out.println(s1 == s2);  System.out.println(s1.equals(s2));  }  }  Output:  False  false |  | | --- | --- |   In the above program, s1 and s2 are having the same data but different references because s1 and s2 are pointing to two different objects.  As references are different == operator will return false. In this case, equals method also returns false because the StringBuffer class has not overridden equals method to compare data. So in case of StringBuffer equals method will get a call from Object class which is meant for reference comparison. So s1.equals(s2) will return false.   | public class Test {  public static void main(String[] args) {  String s1 = new String("sam");  String s2 = new String("sam");  String s3 = "sam";  String s4 = "sam";  }  } |  | | --- | --- |   In above program,  At the time of s1, jvm will create two objects of String with data “sam”, one in pool and one in heap and s1 will point to heap object.  At the time of s2, jvm will create only one object that too in heap area and s2 will point to that object. In this case, the second object in the pool because the pool has already object with data “sam”.  At the time of s3, jvm will create objects in the pool area. But the pool already contains that object so jvm will not create a new object. Simply s3 will point to that object. This will happen for s4 variable too.   | public class Test{  public static void main(String[] args) {  String s = "sam";  s.concat(" abc");  System.out.println(s);  }  }  Output: sam |  | | --- | --- |   In the above program, when we want to perform modification on String objects. Here concat is runtime activity, so jvm will create a new Object in Heap area. But we have not re-assigned concat operation to s, no one will point to newly created objects.   | public class Test{  public static void main(String[] args) {  String s = "sam";  s = s.concat(" abc");  System.out.println(s);  }  }  Output: sam abc |  | | --- | --- |   **String hashcode Cashing:**  String hashcode will calculate only once. And will cashed in our String object.  When we do String s = “sam”, to generate hashcode of this object, jvm will use all the characters s,a,m and will generate hashcode 1010 and will store in object which is pointing by s as given below:    If we want to use this s somewhere in that case jvm will not calculate hashcode again because hashcode is already cashed so time of hashcode calculation will be saved.  **Why is String immutable?**  1. Requirement of String Pool  String pool (String intern pool) is a special storage area in Heap [Area](https://www.programcreek.com/2013/04/jvm-run-time-data-areas/). When a string is created and if the string already exists in the pool, the reference of the existing string will be returned, instead of creating a new object.  For Example:  String string1 = "abcd";  String string2 = "abcd";  If a string is mutable, changing the string with one reference will lead to the wrong value for the other references.  If a string is mutable, changing the string with one reference will lead to the wrong value for the other references.  2. Caching Hashcode  The hashcode of a string is frequently used in Java. For example, in a HashMap or HashSet. Being immutable guarantees that hashcode will always be the same so that it can be cashed without worrying about the changes.That means, there is no need to calculate hashcode every time it is used. This is more efficient.  3. Immutable objects are naturally thread-safe  Because immutable objects can not be changed, they can be shared among multiple threads freely. This eliminates the requirements of doing synchronization.  So, suppose multiple threads are accessing the same string object and when one thread is trying to modify string object, jvm will create new STring object for that thread so that changes will not reflect to other threads.  4. If String is not immutable then it would cause severe security threat to the application. For example, database username, password are passed as String to get database connection and in [socket programming](https://www.journaldev.com/741/java-socket-programming-server-client) host and port details passed as String. Since String is immutable it’s value can’t be changed otherwise any hacker could change the referenced value to cause security issues in the application.  **Why do we prefer String as key in hashmap?**  **In** hashmap, we put data in the form key value. So whenever someone wants to access hashmap data, he will access it by key. So anyone trying to access value with the help of key, first jvm will calculate hashcode of key and then it will search for particular key in hashmap.  But if we use String as a key, we know the hashcode of String object will calculated only once. So when we try to access value with the help of key, jvm will not calculate hashcode every time. Hence calculating hashcode that time we can save and ultimately performance will increase.  **How does Hashmap work internally?**   | public class Test{  public static void main(String[] args) {  HashMap m = new HashMap();  m.put("rollNo", 101);  m.put("name", “sam”);  m.get(“name”);  System.out.println(m);  }  } |  | | --- | --- |   When we create an object of hashmap, jvm will allocate hashtable with 16 buckets. Indexing will start from 0 to 15.  **When we put rollno=101,**  Jvm will create the object of rollno and 101. But where to add these objects in that hashtable. For this jvm will generate hashcode of rollno object and suppose hashcode is 65. It will do 65%16 and we will get 1. So jvm will store that key and value in the second bucket as given in the above diagram.  **When we put name=sam,**  Jvm will create an object of name and sam. But where to add these objects in that hashtable. For this jvm will generate hashcode of name object and suppose hashcode is 35. It will do 35%16 and we will get 3. So jvm will store that key and value in the second bucket as given in the above diagram.  Now while inserting key value, we have to always check for duplicate keys. So to find duplicate keys, jvm will use the equals method internally. In case of duplicate keys, we will not get errors instead of that jvm will replace old value with value of that key.  So to insert a key value pair in hashmap, jvm will use hashcode and equals method internally.  **Retrieving value from Hashmap:**  **While** accessing value from Hashmap, we use key. So when use use suppose m.get(“name”),  Jvm will calculate the hashcode of the name object and according to that hashcode, jvm will go to a particular bucket and there may be a chance that that bucket is having multiple key value pairs. So to access the exact key value pair, jvm uses the equals method to compare key data and if jvm get key then it will return its value.  So, while accessing value from Hashmap, Jvm will use both hashcode and equals method. Hashcode method to find particular bucket and equals method to find particular key.  **How null key is handled in HashMap? Since equals() and hashCode() are used to store and retrieve values, how does it work in case of the null key?**  The null key is handled specially in HashMap, there are two separate methods for that putForNullKey(V value) and getForNullKey(). Later is an offloaded version of get() to look up null keys.  Null keys always map to index 0.  This null case is split out into separate methods for the sake of performance in the two most commonly used operations (get and put), but incorporated with conditionals in others. In short, equals() and hashcode() methods are not used in case of null keys in HashMap.  **What will happen if two different objects have the same hashcode?**  **In the case,** When two key objects are having same hashcode, both the key value pairs will get store in the same bucket as given below diagram:    **How will you retrieve a Value object if two Keys will have the same hashcode?**  Two keys are having the same hashcode that means both the key value pairs will get stored in the same bucket of Hashtable.  While accessing value using key, first jvm will calculate hashcode of that key and will go to respective bucket but there will be two key value pairs so which one is requested one, to select particular key jvm will use equals method.  Using equals method, where we find a match of key, jvm will return that key value pair.  **Can we use any custom object as a key in HashMap?**  This is an extension of previous questions. Of course you can use any Object as key in Java HashMap provided it follows equals and hashCode to be override in order to get correct working and its hashCode should not vary once the object is inserted into [Map](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html). If the custom object is Immutable then this will be already taken care because you can not change it once created.   | public class Employee {  private int id;  private String name;  private int sal;  public Employee(int id, String name, int sal) {  super();  this.id = id;  this.name = name;  this.sal = sal;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getSal() {  return sal;  }  public void setSal(int sal) {  this.sal = sal;  }  @Override  public String toString() {    return "["+this.getId()+" "+this.getName()+" "+this.getSal()+"]";  }  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + id;  result = prime \* result + ((name == null) ? 0 : name.hashCode());  result = prime \* result + sal;  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  Employee other = (Employee) obj;  if (id != other.id)  return false;  if (name == null) {  if (other.name != null)  return false;  } else if (!name.equals(other.name))  return false;  if (sal != other.sal)  return false;  return true;  }} | import java.util.HashMap;  public class Test{  public static void main(String[] args) {  Employee e1 = new Employee(101, "Sam", 100);  Employee e2 = new Employee(101, "Sam", 100);  Employee e3 = new Employee(102, "ram", 300);  Employee e4 = new Employee(103, "Shyam", 400);  HashMap<Employee, Employee> m = new HashMap<Employee, Employee>();  m.put(e1, e1);  m.put(e2, e2);  m.put(e3, e3);  m.put(e4, e4);  System.out.println(m.values());  }  } | | --- | --- |   In the above example, we have four Employee objects among which e1 and e2 are having the same data. Now suppose we don’t override equals and hashcode method, jvm will treat both objects as different objects because both are having different hashcode, even though they are having the same data and jvm will insert both the object.  To overcome this problem, I have override equals and hashcode methods in the above program. |
| **Java 8 Features**  **Java 8 has following advantages:**   1. **Concise code: Java** allows us concise code because of functional programming which is added in Java 8. Concise code means we can do complex task with very small code. 2. **Lambda Expression: It** enabled functional programming.  | **Without Lambda**  public class Test {  public static int square(int i){  return i\*i;  }  public static void main(String[] args) {  System.out.println("4\*4= "+square(4));  System.out.println("5\*5= "+square(5));  }  } | **WIth Lambda**  import java.util.function.Function;  public class Test {    public static void main(String[] args) {  Function<Integer, Integer> f = i->i\*i;  System.out.println("4\*4= "+f.apply(4));  System.out.println("5\*5= "+f.apply(5));  }  } | | --- | --- |   **Java 8 features:**   1. **Lambda Expression** 2. **Functional Interface** 3. **Default and static methods** 4. **Predefined functional interfaces** 5. **Double colon operator(::)** 6. **Streams** 7. **Date and Time api** 8. **Optional class** 9. **Nashorn Javascript Engine**   **Lambda Expression:**  The main objective of lambda expression is to bring functional programming in java.  Functional programming allows us to write complex task in a simple way.  **What is lambda Expression?**  It is an anonymous function.  An Anonymous function is a function without name,return type and modifier.   | Public void m1(){  System.out.println(“Hello”);  } | ()->{System.out.println(“Hello”);}  **If lambda expression body contains only one statement then curly brackets are optional**  **() -> System.out.println(“Hello”);** | | --- | --- | | Public void m1(int a,int b){  System.out.println(a+b);  } | (a,b) -> System.out.println(a+b); | | Public int square(int n){  Return n\*n;  } | (int n) ->{return n\*n;}  Or  (int n) -> n\*n;  (if we don’t write a curly bracket then return keyword is not allowed.  Or  (n) -> n\*n;  Or  If the method has only one parameter then parenthesis are also optional.  So,  n->n\*n; | | Public int m1(String s){  Return s.length();  } | (String s) -> {return s.length();}  Or  (s) -> s.length();  Or  s->s.length(); |   **Rules regarding return statement:**   | n->return n\*n; ==>invalid | Without a curly bracket we can not use the return keyword. | | --- | --- | | n->{return n\*n;}; ⇒ valid |  | | n->{return n\*n}; ⇒ Invalid | semicolon missing | | n->{n\*n;}; ==>Invalid | If we are using a curly bracket and we are returning something then return keyword is mandatory. | | n->n\*n; ==>valid |  |   **Note:**   1. Without a curly bracket we can not use the return keyword. Compiler will automatically consider returned value. 2. Within the curly bracket if we want to return any value, we need to provide a return keyword.   Once we write a lambda expression, we need Functional Interface to call that expression.  **Functional Interface:**  The interface which contains a single abstract method(SAM) is called the Functional Interface.  Ex:  Runnable => run()  Comparable =>compareTo()  Comparator =>compare()  Callable => call()  ActionListener => actionPerformed()  An Functional Interface can have only one abstract method but any number of default and static methods. That restriction is only on abstract methods.     | Interface I{  Public void m1();  Default void m2(){  }  Public static void m3(){  }  } | | --- |   In the above program, I interface is having exactly one abstract method, still it is not a functional interface.  If we want to tell compiler that I is a functional interface then we need to use @FunctionalInterface annotation as given below:   | **@FunctionalInterface**  Interface I{  Public void m1();  Default void m2(){  }  Public static void m3(){  }  } | | --- |  | **@FunctionalInterface**  Interface I{  Default void m2(){  }  Public static void m3(){  }  } | Here, we will get a compile time error because FunctionalInterface should have one abstract method.  And I interface doesn't have an abstract method and we are specifying I as a Functional interface using **@FunctionalInterface** annotation. | | --- | --- | | **@FunctionalInterface**  Interface A{  Public void m1();  }  }  **@FunctionalInterface**  Interface B implements A{  }  } | Here, it is valid. Because B interface doesn't have an abstract method but B got abstract method from A. so it is valid. | | **@FunctionalInterface**  Interface A{  Public void m1();  }  }  **@FunctionalInterface**  Interface B implements A{  Public void m2();  }  } | This is invalid because B is a functional interface and it can not have multiple abstract methods. | | **@FunctionalInterface**  Interface A{  Public void m1();  }  }  Interface B implements A{  Public void m2();  }  } | This is valid because B is not a functional interface.  FUnctional Interface can have non functional children. | | **@FunctionalInterface**  Interface A{  Public void m1();  }  }  **@FunctionalInterface**  Interface B implements A{  Public void m1();  }  } | This is valid because B has overridden m1 method. So, we can not say B has two abstract methods.  Basically B has only one method m1, hence it is a functional interface. | | interface A{  public void m1();  }  ---------------------------------------------  class Demo implements A{  @Override  public void m1() {  System.out.println("M1 method impl");  }    }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  A a = new Demo();  a.m1();  }  } | interface A{  public void m1();  }  public class Test {    public static void main(String[] args) {  A a = ()->System.out.println("M1 method impl using lambda");;  a.m1();  }  }  ---------------------------------------------  Here, A interface should have only one abstract method that is A should be Functional Interface. | | interface A{  public void add(int a,int b);  }  ---------------------------------------------  class Demo implements A{  @Override  public void add(int a,int b) {  System.out.println(a+b);  }    }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  A a = new Demo();  a.add(10,20);  }  } | interface A{  public void add(int a,int b);  }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  A a = (x,y)->System.out.println(x+y);  a.add(10,20);  }  }  ---------------------------------------------  At highlighted location x and y don’t have any type. Compiler will automatically understand those are int type by looking at parent interface A. |   **Note: Compiler will not generate separate .class file for lambda expression.**  **Usage of Lambda Expression:**   1. **In Multithreading**  | **Without Lambda**  class MyThread implements Runnable{  @Override  public void run() {  for(int i=0;i<10;i++){  System.out.println("New Thread");  }  }  }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  MyThread m = new MyThread();  Thread t = new Thread(m);  t.start();  for(int i=0;i<10;i++){  System.out.println("Main Thread");  }  }  } | **With Lambda**  public class Test {    public static void main(String[] args) {  Runnable m = ()->{  for(int i=0;i<10;i++){  System.out.println("New Thread");  }  };  Thread t = new Thread(m);  t.start();  for(int i=0;i<10;i++){  System.out.println("Main Thread");  }  }  } | | --- | --- |   **2. In Collection**   | **Without Lambda**  import java.util.Comparator;  import java.util.TreeSet;  class MyComparator implements Comparator<Integer>{  @Override  public int compare(Integer I1, Integer I2) {    return -(I1.compareTo(I2));  }    }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  TreeSet<Integer> t = new TreeSet<Integer>(new MyComparator());  t.add(10);  t.add(7);  t.add(4);  t.add(13);  t.add(11);  t.add(5);  System.out.println(t);  }  } | **With Lambda Expression**  import java.util.Comparator; import java.util.TreeSet;   public class Test {    public static void main(String[] args) {  Comparator<Integer> c = (I1,I2)->{return -(I1.compareTo(I2));};  TreeSet<Integer> t = new TreeSet<Integer>(c);  t.add(10);  t.add(7);  t.add(4);  t.add(13);  t.add(11);  t.add(5);  System.out.println(t);  } } | | --- | --- | | **Without Lambda Expression**  class MyComparator implements Comparator<Integer>{  @Override  public int compare(Integer I1, Integer I2) {  return I1.compareTo(I2);  }    }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  ArrayList<Integer> t = new ArrayList<Integer>();  t.add(10);  t.add(7);  t.add(4);  t.add(13);  t.add(11);  t.add(5);  t.add(10);  Collections.sort(t,new MyComparator());  System.out.println(t);  }  } | **With Lambda Expression**  public class Test {    public static void main(String[] args) {  Comparator<Integer> c = (I1,I2)->{return -(I1.compareTo(I2));};  ArrayList<Integer> t = new ArrayList<Integer>();  t.add(10);  t.add(7);  t.add(4);  t.add(13);  t.add(11);  t.add(5);  t.add(10);  Collections.sort(t,c);  System.out.println(t);  }  } | | class Employee{  int id;  String name;  public Employee() {  }  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }  }  ---------------------------------------------  class MyComp implements Comparator<Employee>{  @Override  public int compare(Employee e1, Employee e2) {  Integer I1 = e1.id;  Integer I2 = e2.id;  return -(I1.compareTo(I2));  }    }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  Employee e1 = new Employee(104, "shyam");  Employee e2 = new Employee(101, "sam");  Employee e3 = new Employee(103, "ram");  Employee e4 = new Employee(102, "ganesh");  ArrayList<Employee> al = new ArrayList<Employee>();  al.add(e1);  al.add(e2);  al.add(e3);  al.add(e4);  Collections.sort(al,new MyComp());  for (int i = 0; i < al.size(); i++) {  System.out.println(al.get(i).id+" "+al.get(i).name);  }  }  } | import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  class Employee{  int id;  String name;  public Employee() {  }  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }  }  ---------------------------------------------  public class Test {    public static void main(String[] args) {  Employee e1 = new Employee(104, "shyam");  Employee e2 = new Employee(101, "sam");  Employee e3 = new Employee(103, "ram");  Employee e4 = new Employee(102, "ganesh");  ArrayList<Employee> al = new ArrayList<Employee>();  al.add(e1);  al.add(e2);  al.add(e3);  al.add(e4);  /\*Comparator<Employee> c = (ex,ey)->{  Integer I1 = ex.id;  Integer I2 = ey.id;  return -(I1.compareTo(I2));  };  \*/  Comparator<Employee> c = (ex,ey)->(ex.id>ey.id)?-1:(ex.id<ey.id)?1:0;  Collections.sort(al,c);  for (int i = 0; i < al.size(); i++) {  System.out.println(al.get(i).id+" "+al.get(i).name);  }  }  } |   **Anonymous Inner class vs Lambda Expression:**   | **Anonymous Inner Class**  **--------------------------------------------------**  public class Test {    public static void main(String[] args) {  Runnable r = new Runnable() {    @Override  public void run() {  for (int i = 0; i < 10; i++) {  System.out.println("New Thread");  }  }  };  Thread t = new Thread(r);  t.start();  for (int i = 0; i < 10; i++) {  System.out.println("Main thread");  }  }  } | **Lambda Expression**  **--------------------------------------------------**  public class Test {    public static void main(String[] args) {  Runnable r = ()->{  for (int i = 0; i < 10; i++) {  System.out.println("New thread");  }  };  Thread t = new Thread(r);  t.start();  for (int i = 0; i < 10; i++) {  System.out.println("Main thread");  }  }  } | | --- | --- |  | **Note:** Anonymous Inner class and Lambda Expression are not alternative.  When we are implementing Functional Interface using anonymous inner class in that case only we can replace anonymous inner class with lambda expression. | | --- | | **Anonymous Inner class:**   1. **Anonymous inner class can extend normal as well as abstract class.** 2. **Anonymous class can implement normal as well as functional interface.**   **Lambda Expression:**   1. **Lambda Expression can implement only functional interface.** |   **Default method:**  From Java 8, we can write the default method inside the interface.  **Need:**  Suppose we have Mobile interface as given below:   | interface Mobile{  public void call();  public void sms();  }   | public class Airtel implements Mobile{  @Override  public void call() {    }  @Override  public void sms() {    }    } | public class Vodafone implements Mobile{  @Override  public void call() {    }  @Override  public void sms() {    }    } | | --- | --- |   Here,  Mobile interface is implemented by Airtel and Vodafone because they want call and sms method.  In the future suppose we want to add a videoCall method in the Mobile interface. So if we add videoCall method in Mobile interface, Airtel and Vodafone both must implement videoCall method compulsory.  But I think a situation where Airtel doesn't want videoCall still Airtel has to implement that method. SO this is the problem.  To overcome this problem we can use default method as given below: | | --- | --- | --- | | interface Mobile{  public void call();  public void sms();  default void videoCall(){}  }   | public class Airtel implements Mobile{  @Override  public void call() {    }    @Override  public void sms() {    }    } | public class Vodafone implements Mobile{  @Override  public void call() {    }  @Override  public void videoCall() {  Mobile.super.videoCall();  }  @Override  public void sms() {    }    } | | --- | --- |   Here, the default method has logic. It is not abstract so Airtel class if you don't want then Airtel will not override.  **So,Without affecting implementation classes, we can add new methods into Interface with the help of default methods.** |  | interface I{    default void videoCall(){}  }  public class Test implements I{    @Override  public void videoCall() {    }  } | **If we** observe here, In Test class we have given the public. It should be public only.  We can not write the default keyword in any class because default is reserved word for switch case. | | --- | --- | | interface I{    default int hashCode(){  return 10;  }  }  public class Test implements I{    } | **Here** we will get errors because we can not write Object class method as default.  Because Object class methods are already available to every class then what is the need of making those methods available via default method. |   interface Left{    default void m1(){  System.out.println("Left m1 method");  }  }  interface Right{    default void m1(){  System.out.println("Right m1 method");  }  }  public class Test implements Left,Right{    }  Here, we will get errors because we are implementing both Left and Right and both are having the same method so if we try to access m1 using a child class object then which method should get call. So whenever we are implementing multiple interface which are having same default method then compulsory we should override it as given below:  interface Left{    default void m1(){  System.out.println("Left m1 method");  }  }  interface Right{    default void m1(){  System.out.println("Right m1 method");  }  }  public class Test implements Left,Right{  @Override  public void m1() {  System.out.println("Child Implementation");  }  public static void main(String[] args) {  Test t = new Test();  t.m1();  }  }  **Output: Child Implementation**  **But what if we want to call super interface methods? In that case use following code:**  public class Test implements Left,Right{  @Override  public void m1() {  **Left.super.m1();//OR**  **Right.super.m1();**  }  public static void main(String[] args) {  Test t = new Test();  t.m1();  }  } |
| **Static method:**  From Java 8 onwards, we can write static methods inside the interface.  When we want to write some logic and we don’t want to deal with any object creation then instead of writing class, write interface and use static method in that interface as given below:   | interface I{  public static void m1(){  System.out.println("Interface static method");  }  }  **--------------------------------------------------**  public class Test implements I{    public static void main(String[] args) {  I.m1();  //m1(); --- 1  //Test.m1(); --- 2  //new Test().m1(); --- 3  }  } | We can access the interface static method using only the interface name itself.  We can not access interface static methods using child class, so in this program, statement 1,2 and 3 are wrong leads to compile time error. | | --- | --- | | interface I{  public static void main(String[] args) {  System.out.println("Main method inside interface");  }  }  **Output : Main method inside interface** | We can also use the main method inside the interface. |   **What is the need for a static method inside the interface?**  Suppose, we want to write some utility methods. All utility methods are static and we don’t have any instance variable then we can use interface. Because if we use class to declare those static utility methods, performance will degrade because class deals with constructor call and all.  **So, we can use interfaces with static methods to declare static utility methods.** |
| **Predefined Functional Interfaces:**  There are few predefined functional interface which are given below:   1. **Predicate** 2. **Function** 3. **Consumer** 4. **Supplier**   There are few two parameter functional interfaces which are given below:   1. **BiPredicate** 2. **BiFunction** 3. **BiConsumer**   There are few primitive functional interfaces which are given below:   1. **IntPredicate** 2. **IntFunction** 3. **IntConsumer**   **Predicate:**  It is used for conditional checks.  As it is functional interface, it contains only one abstract method which is given below:  **Public abstract boolean test(T t);**   | **Without Predicate**  public class Test{    public boolean test(Integer I){  if(I%2 == 0){  return true;  }else{  return false;  }  }  public static void main(String[] args) {  Test t = new Test();  System.out.println(t.test(10));  System.out.println(t.test(15));  }  } | **With Predicate**  public class Test{  public static void main(String[] args) {  Predicate<Integer> p = (Integer I)->I%2==0;  System.out.println(p.test(10));  System.out.println(p.test(15));  }  }  Or  **--------------------------------------------------**  public class Test{  public static void main(String[] args) {  Predicate<Integer> p = I->I%2==0;  System.out.println(p.test(10));  System.out.println(p.test(15));  }  } | | --- | --- | | class Employee{  int id;  String name;  public Employee() {  }  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }    }  **--------------------------------------------------**  public class Test{  public static void main(String[] args) {  Employee e1 = new Employee(101, "sam");  Employee e2 = new Employee(102, "ram");  Predicate<Employee> p = (e)->e.id<102;  System.out.println(p.test(e1));  System.out.println(p.test(e2));  }  } | class Employee{  int id;  String name;  public Employee() {  }  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }    }  **-------------------------------------------------**  public class Test{  public static void main(String[] args) {  Employee e1 = new Employee(101, "sam");  Employee e2 = new Employee(102, "ram");  Predicate<Employee> p = (e)->e.id==102 || e.name.equals("sam");  System.out.println(p.test(e1));  System.out.println(p.test(e2));  }  } | | public class Test{  public static void main(String[] args) {  String[] str = {"sam","ram","shyam","ganesh"};  Predicate<String> p = s->s.length()==3;    for(String s1:str){  if (p.test(s1)) {  System.out.println(s1);  }  }  }  } | import java.util.ArrayList;  import java.util.function.Predicate;  class Employee{  int id;  String name;  int sal;  public Employee() {  }  public Employee(int id, String name,int sal) {  super();  this.id = id;  this.name = name;  this.sal = sal;  }    }  **--------------------------------------------------**  public class Test{  public static void main(String[] args) {  ArrayList<Employee> al = new ArrayList<Employee>();  al.add(new Employee(101, "sam",2345));  al.add(new Employee(102, "ram",1000));  al.add(new Employee(103, "shyam",1234));  al.add(new Employee(104, "sameer",500));  al.add(new Employee(105, "rohan",700));  Predicate<Employee> p = e->e.sal>=1000;  for(Employee e:al){  if (p.test(e)) {  System.out.println(e.id+" "+e.name+" "+e.sal);  }  }  }  } |   **Function Functional Interface:**  It is a functional interface which has following abstract method:  **Public R apply(T,R);**  **Here,**  R can be any type. Because apply method returns a single value of any type.   | public class Test{  public static void main(String[] args) {  Function<String, Integer> f = s->s.length();  System.out.println(f.apply("Sam"));  System.out.println(f.apply("Rohit"));  }  } | Here, Function<String, Integer> means lambda expression will consume String value and it will return Integer value.  **Output:**  **3**  **5** | | --- | --- | | public class Test{  public static void main(String[] args) {  Function<Integer, Integer> f = I->I\*I;  System.out.println(f.apply(5));  System.out.println(f.apply(6));  }  } | Function<Integer, Integer> means lambda expression will consume Integer value and will return Integer value. | | import java.util.function.Function;  class Student{  int rn;  String name;  int marks;  public Student(int rn, String name, int marks) {  this.rn = rn;  this.name = name;  this.marks = marks;  }    }  **--------------------------------------------------**  public class Test{  public static void main(String[] args) {    Student s1 = new Student(101, "Sam", 57);  Student s2 = new Student(102, "Ram", 27);  Student s3 = new Student(103, "Shyam", 37);  Student s4 = new Student(104, "Sameer", 67);  Function<Student, String> f = s->{  String grade=null;  if(s.marks>=35)grade="pass";  else grade ="fail";  return grade;  };  System.out.println(f.apply(s1));  System.out.println(f.apply(s2));  System.out.println(f.apply(s3));  System.out.println(f.apply(s4));  }  } | public class Test{  public static void main(String[] args) {    Function<Integer, Integer> f1 = I->2\*I;  Function<Integer, Integer> f2 = I->I\*I\*I;    System.out.println(f1.apply(2));  System.out.println(f2.apply(2));  System.out.println(f1.andThen(f2).apply(2));  System.out.println(f1.compose(f2).apply(2));  }  }  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **Here,**  f1.andThen(f2).apply(2) means  First f1 will get a call with value 2 and output will return as 4. Now this 4 will pass to f2 and finally we will get 4\*4\*4 as 64.  f1.compose(f2).apply(2) means  F2 will get a call first with value 2 and will get returned output 8. Now this 8 value will pass to f1 so finally we will get output as 16.  Above both situations are called Function **chaining.** |   **Consumer Functional Interface:**  This is an functional interface which has following method:  **Public void accept(T t);**  **Function** interface consumes as well as returns value whereas **Consumer** only consumes data. It does not return anything.   | public class Test{  public static void main(String[] args) {  Consumer<String> c = s->System.out.println(s);  c.accept("This is consumer interface");  }  } | Here,  Consumer interface only consumes. Does not return anything.  In this example, it consumes String data.  Return type of Consumer is void. | | --- | --- |  | import java.util.ArrayList;  import java.util.function.Consumer;  class Student{  int rn;  String name;  int marks;  public Student(int rn, String name, int marks) {  this.rn = rn;  this.name = name;  this.marks = marks;  }    }  public class Test{  public static void main(String[] args) {  ArrayList<Student> al = new ArrayList<Student>();  al.add(new Student(101, "Sam", 57));  al.add(new Student(102, "Ram", 27));  al.add(new Student(103, "Shyam", 37));  al.add(new Student(104, "Sameer", 67));    Consumer<Student> c = s->{  System.out.println(s.rn+" "+s.name+" "+s.marks);  };    for(Student s : al){  c.accept(s);  }    }  } | | --- |   **Supplier Functional Interface:**  Here It supplies required objects and it will not take any input.  Supplier interface has following method:  **Public R get();**   | public class Test{    public static void main(String[] args) {  Supplier<Date> d = ()->new Date();  System.out.println(d.get());  }  } | **Here,** Supplier supplies Date class objects. | | --- | --- | | public class Test{    public static void main(String[] args) {  Supplier<String> d = ()->{  String otp = "";  for (int i = 0; i < 6; i++) {  otp = otp + (int)(Math.random()\*10);  }  return otp;    };  System.out.println(d.get());  }  } | **This example we can use as OTP generation.** |   **Two Augmented Functional interfaces:**   1. **BiPredicate: This** is a functional interface which accepts two parameters. For example, when we want to check the condition for two values, we can use BiPredicate.  | public class Test{    public static void main(String[] args) {  BiPredicate<Integer, Integer> b = (x,y)->(x+y)%2==0;  System.out.println(b.test(10, 20));  System.out.println(b.test(10, 5));  }  } | Here, BiPredicate accepts two values of type Integer and returns boolean value. | | --- | --- | | class Employee{  int id;  String name;  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }    }  **--------------------------------------------------**  public class Test{    public static void main(String[] args) {  BiPredicate<Employee, Employee> p = (e1,e2)->e1.name.equals(e2.name);  System.out.println(p.test(new Employee(101, "sam"), new Employee(102, "ram")));  System.out.println(p.test(new Employee(103, "shyam"), new Employee(104, "shyam")));  }  } | Here, BiPredicate accepts two objects of an employee and returns true if both employees are having the same name otherwise return false. |   **2. BiFunctional Interface**  This is a functional interface which accepts two parameters and returns desired result.   | class Employee{  int id;  String name;  public Employee(int id, String name) {  super();  this.id = id;  this.name = name;  }    }  **--------------------------------------------------**  public class Test{    public static void main(String[] args) {  BiFunction<Integer, String, Employee> f = (id,name)->new Employee(id, name);  Employee e1 = f.apply(101, "sam");  Employee e2 = f.apply(102, "ram");  System.out.println("First Employee: ");  System.out.println(e1.id+" "+e1.name);  System.out.println("Second Employee: ");  System.out.println(e2.id+" "+e2.name);  }  } | Here, BiFunction interface access two values of type Integer and String and returns Employee object. | | --- | --- |   **3.BiConsumer:**  It is a functional interface which accepts two values and does not return anything.   | class Employee{  int id;  String name;  int sal;  public Employee(int id, String name,int sal) {  super();  this.id = id;  this.name = name;  this.sal = sal;  }    }  public class Test{  public static void populate(ArrayList<Employee> al){  al.add(new Employee(101, "sam", 1000));  al.add(new Employee(102, "ram", 2000));  al.add(new Employee(103, "shyam", 3000));  }  public static void main(String[] args) {  ArrayList<Employee> al = new ArrayList<Employee>();  populate(al);  System.out.println("Before Bonus");  for(Employee e:al){  System.out.println(e.id+" "+e.name+" "+e.sal);  }  BiConsumer<Employee, Integer> c = (e,bonus)->e.sal = e.sal+bonus;  for(Employee e:al){  c.accept(e, 100);  }  System.out.println("-------------------");  System.out.println("After Bonus");  for(Employee e:al){  System.out.println(e.id+" "+e.name+" "+e.sal);  }  }  } | Here BiConsumer takes two values Employee object and integer and adds a given bonus into everyone's salary.   | public class Test{  public static void main(String[] args) {  Predicate<Integer> p = i->i%2==0;  System.out.println(p.test(4));  }  } | | --- |   If we observe the above example, we are passing primitive value 4. But while storing into i, it will convert into an Integer object and when we apply % operator again it convert from Integer object to int value. | | --- | --- | --- |   So, here a lot of autoboxing and autounboxing. To overcome this problem, we have primitive functional interfaces.   1. **IntPredicate** 2. **LongPredicate** 3. **DoublePredicate**   **4. DoubleFunction:** It can accept double as input and return type can be anything.  **5. IntFunction:** It can accept int as input  **6. LongFunction:** It can accept long as input  **7.DoubleToIntFunction:** It can accept double value and it will return int value.  **8.DoubleToLongFunction:** It can accept double value and will return long value.  **9.IntToDoubleFunction:** It can accept int as input and will return double value.  **10.IntToLongFunction:** It can accept int value as input and will return Long as output.  **11. LongToIntFunction:** It takes long input and returns int value.  **12. LongToDoubleFunction:** It takes long input and returns double value.  **13.ToIntFunction:** It takes any type of input but it will return int value.  **14.ToLongFunction:** It takes any type of input but it will return long value.  **15.ToDoubleFunction:** It takes any type of input but it will return double value.  **16. ToIntBiFunction:** It takes any type of two parameters and returns int value.  **17. ToLongBiFunction:** It takes any type of two parameters and returns long value.  **18. ToDoubleBiFunction:** It takes any type of two parameters and returns double value.  **19. IntConsumer:** It takes int value as input but does not return anything.  **20. LongConsumer:** It takes long value as input but does not return anything.  **21. DoubleConsumer:** It takes double value as input but does not return anything.  **22. ObjDoubleConsumer:** It takes two arguments first can be anything and second is double and it does not return anything.  **23. ObjIntConsumer:** It takes two arguments first can be anything and second is int and it does not return anything.  **24. ObjLongConsumer:** It takes two arguments first can be anything and second is long and it does not return anything.  **25.BooleanSupplier:** It supplies boolean value.  **26.IntSupplier:** It supplies int value.  **27. LongSupplier:** It supplies long value. |
| **Method Reference**  Consider following example of lambda expression:   | public class Test{  public static void main(String[] args) {  Runnable r = ()->{  for(int i =0;i<10;i++){  System.out.println("New Thread");  }  };  Thread t = new Thread(r);  t.start();  for(int i =0;i<10;i++){  System.out.println("Main Thread");  }  }  } | | --- |   Here in this example whatever we have written in the run method of Runnable, we can not reuse any where.  Instead of this, we can write separate method which contains that logic and we can call that method using method reference concept as given below:   | public class Test{  public static void m1(){  for(int i =0;i<10;i++){  System.out.println("New Thread");  }  }  public static void main(String[] args) {  Runnable r = Test::m1;  Thread t = new Thread(r);  t.start();  for(int i =0;i<10;i++){  System.out.println("Main Thread");  }  }  } | | --- |   Here in above program, we have declared m1 method as static if it is non static then we will get error because we can not access non static method from static method directly then do following code:   | public class Test{  public void m1(){  for(int i =0;i<10;i++){  System.out.println("New Thread");  }  }  public static void main(String[] args) {  Test t1 = new Test();  Runnable r = t1::m1;  Thread t = new Thread(r);  t.start();  for(int i =0;i<10;i++){  System.out.println("Main Thread");  }  }  } | interface I{  public void add(int a,int b);  }  public class Test{  public static void sum(int x,int y){  System.out.println(x+y);  }  public static void main(String[] args) {  I i = Test::sum;  i.add(10, 20);  }  } | | --- | --- | |
| **Date and Time API(Joda Time API):**  **Java 8 has** added a new Date and Time API which is also known as Joda Time API.  It is given by joda.org people, hence it is called Joda Time API.   | import java.time.LocalDate;  import java.time.LocalTime;  public class Test{  public static void main(String[] args) {  LocalDate date = LocalDate.now();  System.out.println("Date: "+date);  LocalTime time = LocalTime.now();  System.out.println("Time: "+time);  }  } | public class Test{  public static void main(String[] args) {  LocalTime time = LocalTime.now();  System.out.println("Hour: "+time.getHour());  System.out.println("Minute: "+time.getMinute());  System.out.println("Second: "+time.getSecond());  System.out.println("NanoSeconds: "+time.getNano());  }  } | | --- | --- | | public class Test{  public static void main(String[] args) {  LocalDate date = LocalDate.now();  System.out.println("Day: "+date.getDayOfMonth());  System.out.println("Month: "+date.getMonth());  System.out.println("Month: "+date.getMonthValue());  System.out.println("Year: "+date.getYear());    }  } | public class Test{  public static void main(String[] args) {  LocalDateTime time = LocalDateTime.now();  System.out.println("DateAndTime: "+time);  System.out.println("Day: "+time.getDayOfMonth());  System.out.println("Hour: "+time.getMinute());  }  } | | public class Test{  public static void main(String[] args) {  LocalDateTime time = LocalDateTime.of(1992, 5, 30, 12, 24);  System.out.println(time);  System.out.println("After Six Months: "+time.plusMonths(6));  System.out.println("After two hour: "+time.plusHours(2));  System.out.println("After ten Min: "+time.plusMinutes(10));  System.out.println("After three days: "+time.plusDays(3));  }  } | public class Test{  public static void main(String[] args) {  LocalDate bdate = LocalDate.of(1992, 5, 30);  LocalDate today = LocalDate.now();  Period period1 = Period.between(bdate, today);  Period period2 = Period.from(period1);  System.out.println(period1.getMonths()+" "+period1.getYears()+" "+period1.getDays());  System.out.println(period2.getMonths()+" "+period2.getYears()+" "+period2.getDays());  }  } | | public class Test{  public static void main(String[] args) {  Scanner sc = new Scanner(System.in);  System.out.println("Enter year here");  int n = sc.nextInt();  Year year = Year.of(n);  if (year.isLeap()) {  System.out.println(year+" is a leap year");  } else {  System.out.println(year+" is not leap year");  }  }  } | public class Test{  public static void main(String[] args) {  ZoneId zoneId = ZoneId.systemDefault();  System.out.println(zoneId);  }  } | |
| **Java 8 Stream** Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allow functional-style operations on the elements. You can use stream by importing java.util.stream package.  Stream provides following features:   * Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations. * Streams are functional in nature. Operations performed on a stream do not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.  | class Test{  public static void main(String[] args) {  List<String> al = Arrays.asList("Sam","Ram","Shyam");    al.stream().filter(name->!name.equals("Sam")).forEach(name->System.out.println(name));  }  } | Here,  **Stream** method will create one stream and it will provide all the elements from al list one by one.  **Filter** method will filter that stream depending on condition given by us.  **forEach** will print the names given by filter. | | --- | --- | | class Test{  public static void main(String[] args) {  ArrayList<Integer> al = new ArrayList<Integer>();  for (int i = 0; i <= 10; i++) {  al.add(i);  }  al.stream().filter(I->I%2==0).forEach(I->System.out.println(I));  }  } | Here, the Stream method will provide all the values from 0 to 10 to filter method. Filter method will filter by given condition and filtered elements will pass to forEach method to print. |   Now Suppose we have one list and we want to check even numbers and want to add only even numbers from that list in new list then do in following way:   | class Test{  public static void main(String[] args) {  ArrayList<Integer> al = new ArrayList<Integer>();  ArrayList<Integer> filteredList = new ArrayList<Integer>();  for (int i = 0; i <= 10; i++) {  al.add(i);  }  al.stream().filter(I->I%2==0).forEach(I->filteredList.add(I));  for(Integer I:filteredList){  System.out.println(I+"..");  }  }  } | | --- |   But here we are creating a new List and then manually we are adding even numbers into it. We can reduce this code as given below:   | class Test{  public static void main(String[] args) {  ArrayList<Integer> al = new ArrayList<Integer>();  for (int i = 0; i <= 10; i++) {  al.add(i);  }  List<Integer> filterList = al.stream().filter(I->I%2==0).collect(Collectors.toList());  for(Integer I:filterList){  System.out.println(I);  }  }  } | Here, the Collect method will collect all the filtered elements into list with name filterList .. Here we don’t need to create List object explicitly | | --- | --- |   Now Suppose we have list of names and we want to create Employee object for each name then do it as given below:   | **Without Lambda Expression**  class Employee {  private String name;  private int sal;    public Employee(String name, int sal) {  super();  this.name = name;  this.sal = sal;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getSal() {  return sal;  }  public void setSal(int sal) {  this.sal = sal;  }    }  **--------------------------------------------------**  class Test{  public static void main(String[] args) {  List<String> list = Arrays.asList("sam","ram","shyam","ganesh");  List<Employee> emplist = list.stream().map(new Function<String, Employee>() {  @Override  public Employee apply(String t) {  return new Employee(t,100);  }  }).collect(Collectors.toList());    for(Employee e:emplist){  System.out.println(e.getName()+" "+e.getSal());  }  }    } | **With Lambda Expression**  class Employee {  private String name;  private int sal;    public Employee(String name, int sal) {  super();  this.name = name;  this.sal = sal;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getSal() {  return sal;  }  public void setSal(int sal) {  this.sal = sal;  }    }  **--------------------------------------------------**  class Test{  public static void main(String[] args) {  List<String> list = Arrays.asList("sam","ram","shyam","ganesh");  List<Employee> emplist = list.stream().map((name)-> {return new Employee(name,200);}).collect(Collectors.toList());    for(Employee e:emplist){  System.out.println(e.getName()+" "+e.getSal());  }  }    } | | --- | --- |   Here map method maps Employee into new List. When we want to convert one value into another type then we use map method. Here we are passing string value and the map method is returning Employee objects.  Now Suppose we want to filter name list and depend on filter result we want to map then do it as given below:   | class Employee {  private String name;  private int sal;  //constructor  // getter and setter    }  class Test{  public static void main(String[] args) {  List<String> list = Arrays.asList("sam","ram","shyam","ganesh");  List<Employee> emplist = list.stream().filter(name->!name.equals("ram")).map((name)-> {return new Employee(name,200);}).collect(Collectors.toList());    for(Employee e:emplist){  System.out.println(e.getName()+" "+e.getSal());  }  }    } | Here filter method will filter the name list and it will not keep ram, so map method will create Employee objects for all the names except ram. | | --- | --- |   Now Suppose we have list of Employee objects and we want to print all the salaries or we want to sum salaries of all the employees then:   | class Employee {  private String name;  private int sal;  //constrictor  // setter and getter    }  **--------------------------------------------------**  class Test{  public static void main(String[] args) {  ArrayList<Employee> al = new ArrayList<Employee>();  al.add(new Employee("Sam", 100));  al.add(new Employee("Ram", 200));  al.add(new Employee("Samantha", 300));  al.add(new Employee("Shyam", 400));  al.add(new Employee("Sameer", 500));    al.stream().mapToInt(emp->emp.getSal()).forEach(s->System.out.println(s));  }  } | class Employee {  private String name;  private int sal;  //constrictor  // setter and getter    }  **--------------------------------------------------**  class Test{  public static void main(String[] args) {  ArrayList<Employee> al = new ArrayList<Employee>();  al.add(new Employee("Sam", 100));  al.add(new Employee("Ram", 200));  al.add(new Employee("Samantha", 300));  al.add(new Employee("Shyam", 400));  al.add(new Employee("Sameer", 500));    int sum = al.stream().mapToInt(emp->emp.getSal()).sum();  System.out.println(sum);  }  } | | --- | --- |  | class Test{  public static void main(String[] args) {  List<String> names = Arrays.asList("sam","ram","shyam","Sameer");  int[] lengths = names.stream().mapToInt(sal->sal.length()).toArray();  for(int a:lengths){  System.out.println(a);  }  }  } | | --- | |
| **Finalize method:**  It is a method that the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) always calls just before the deletion/destroying the object which is eligible for Garbage Collection, so as to perform clean-up activity. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource deallocation. Remember it is not a reserved keyword.  Once the finalize method completes immediately Garbage Collector destroys that object. finalize method is present in Object class and its syntax is:  **protected void finalize throws Throwable{}**  Since Object class contains the finalize method hence finalize method is available for every java class since Object is the superclass of all java classes. Since it is available for every java class hence Garbage Collector can call finalize method on any java object  Now, the finalize method which is present in the Object class, has an empty implementation, in our class clean-up activities are there, then we have to override this method to define our own clean-up activities.  Cases related to finalize method:   1. **Case 1 :** The object which is eligible for Garbage Collection, that object’s corresponding class finalize method is going to be executed   class Hello {  public static void main(String[] args)  {  String s = new String("RR");  s = null;  // Requesting JVM to call Garbage Collector method  System.gc();  System.out.println("Main Completes");  }  // Here overriding finalize method  public void finalize()  {  System.out.println("finalize method overridden");  }  }  **Note** : Here above output came only **Main Completes** and **not** “finalize method overridden” because Garbage Collector calls finalize method on that class object which is eligible for Garbage collection. Here above we have done->**s = null** and ‘s’ is the object of String class, so String class finalize method is going to be called and not our class(i.e, Hello class). So we modify our code like->  Hello s = new Hello();  s = null;  Now our class i.e, Hello class finalize method is called. **Output**:  finalize method overridden  Main Completes  So basically, Garbage Collector calls finalize method on that class object which is eligible for Garbage collection.So if String object is eligible for Garbage Collection then **String** class finalize method is going to be called and **not the Hello class** finalize method.  **Case 2 :** We can call the finalize method Explicitly then it will be executed just like a normal method call but the object won’t be deleted/destroyed.  class Bye {  public static void main(String[] args)  {  Bye m = new Bye();  // Calling finalize method Explicitly.  m.finalize();  m.finalize();  m = null;  // Requesting JVM to call Garbage Collector method  System.gc();  System.out.println("Main Completes");  }  // Here overriding finalize method  public void finalize()  {  System.out.println("finalize method overridden");  }  }  **Output:**  **finalize method overridden**  **//call by the programmer but the object won't get destroyed.**  **finalize method overridden**  **//call by the programmer but the object won't get destroyed.**  **Main Completes**  **finalize method overridden**  **//call by Garbage Collector just before destroying the object.**  Note : As finalize is a method and not a reserved keyword, so we can call finalize method Explicitly, then it will be executed just like a normal method call but object won’t be deleted/destroyed.  **Important points:**   1. There is no guarantee about the time when finalize is called. It may be called any time after the object is not being referred anywhere (can be garbage collected). 2. JVM does not ignore all exceptions while executing finalize method, but it ignores only [Unchecked exceptions](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/). If the corresponding catch block is there then JVM won’t ignore and corresponding catch block will be executed. 3. System.gc() is just a request to JVM to execute the Garbage Collector. It’s up-to JVM to call Garbage Collector or not.Usually JVM calls Garbage Collector when there is not enough space available in the Heap area or when the memory is low. |
| **Garbage Collection in Java:** In C/C++, programmers are responsible for both creation and destruction of objects. Usually programmers neglect the destruction of useless objects. Due to this negligence, at a certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing OutOfMemoryErrors.  But in Java, the programmer needs not to care for all those objects which are no longer in use. Garbage collectors destroy these objects.  Garbage collector is the best example of [Daemon thread](https://www.geeksforgeeks.org/daemon-thread-java/) as it is always running in the background.  Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.  **Unreachable objects :** An object is said to be unreachable if it doesn’t contain any reference to it. Also note that objects which are part of the island [of isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/) are also unreachable.  Integer i = new Integer(4);  // the new Integer object is reachable via the reference in 'i'  i = null;  // the Integer object is no longer reachable.    **Eligibility for garbage collection :** An object is said to be eligible for GC(garbage collection) iff it is unreachable. In the above image, after *i = null;* integer object 4 in heap area is eligible for garbage collection.  **Ways to make an object eligible for GC:**  Even though the programmer is not responsible for destroying useless objects, it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.  There are generally four different ways to make an object eligible for garbage collection.   1. Nullifying the reference variable 2. Re-assigning the reference variable 3. Object created inside method 4. Island of Isolation   Once we make an object eligible for garbage collection, it may not be destroyed immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect it.  We can also request JVM to run Garbage Collector. There are two ways to do it :   1. **Using *System.gc()* method** : System class contains static method *gc()* for requesting JVM to run Garbage Collector. 2. **Using *Runtime.getRuntime().gc()* method** : [Runtime class](https://www.geeksforgeeks.org/java-lang-runtime-class-in-java/) allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.   // Java program to demonstrate requesting  // JVM to run Garbage Collector  public class Test  {  public static void main(String[] args) throws InterruptedException  {  Test t1 = new Test();  Test t2 = new Test();    // Nullifying the reference variable  t1 = null;    // requesting JVM for running Garbage Collector  System.gc();    // Nullifying the reference variable  t2 = null;    // requesting JVM for running Garbage Collector  Runtime.getRuntime().gc();    }  @Override  // finalize method is called on object once  // before garbage collecting it  protected void finalize() throws Throwable  {  System.out.println("Garbage collector called");  System.out.println("Object garbage collected : " + this);  }  }  output:  Garbage collector called  Object garbage collected : Test@46d08f12  Garbage collector called  Object garbage collected : Test@481779b8   * **Note :**   1. There is no guarantee that any one of the above two methods will definitely run Garbage Collector.   2. The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()*   **Finalization**   * Just before destroying an object, Garbage Collector calls the finalize*()* method on the object to perform cleanup activities. Once the finalize*()* method completes, Garbage Collector destroys that object. *finalize()* method is present in [Object class](https://www.geeksforgeeks.org/object-class-in-java/) with the following prototype. protected void finalize() throws Throwable * Based on our requirement, we can override the finalize*()* method for performing our cleanup activities like closing connection from database. * **Note :**   1. The *finalize()* method is called by Garbage Collector not [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). Although Garbage Collector is one of the modules of JVM.   2. [Object class](https://www.geeksforgeeks.org/object-class-in-java/) *finalize()* method has empty implementation, thus it is recommended to override *finalize()* method to dispose of system resources or to perform other cleanup.   3. The *finalize()* method is never invoked more than once for any given object.   4. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored and finalization of that object terminates.   //Correct code to count number  //of employees excluding interns.  class Employee  {  private int ID;  private String name;  private int age;  private static int nextId=1;  //it is made static because it  // is keep common among all and  // shared by all objects  public Employee(String name,int age)  {  this.name = name;  this.age = age;  this.ID = nextId++;  }  public void show()  {  System.out.println  ("Id="+ID+"\nName="+name+"\nAge="+age);  }  public void showNextId()  {  System.out.println  ("Next employee id will be="+nextId);  }  protected void finalize()  {  --nextId;  //In this case,  //gc will call finalize()  //for 2 times for 2 objects.  }  }  //it is closing brace of Employee class  class UseEmployee  {  public static void main(String []args)  {  Employee E=new Employee("GFG1",56);  Employee F=new Employee("GFG2",45);  Employee G=new Employee("GFG3",25);  E.show();  F.show();  G.show();  E.showNextId();  F.showNextId();  G.showNextId();    {  //It is sub block to keep  // all those interns.  Employee X=new Employee("GFG4",23);  Employee Y=new Employee("GFG5",21);  X.show();  Y.show();  X.showNextId();  Y.showNextId();  X = Y = null;  System.gc();  System.runFinalization();  }  E.showNextId();  }  } |