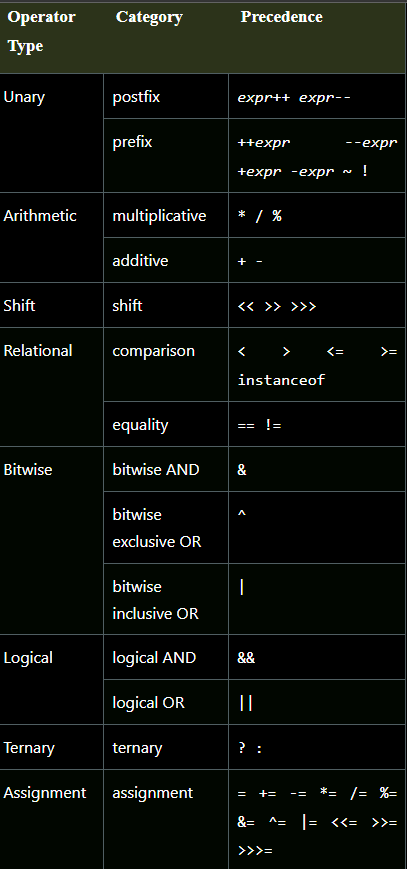
Variables:

Local,instance and static variables.

->a variable declared inside body of the method is called local variable. Other methods in this class aren’t even aware of this variable. (A local variable cannot be defined with “static” key word).

-> A variable declared inside class but out outside body of method without static key word is called instance variable. (because its value is instance -specific and is not shared among instances(objects).

-> A variable that is declared as static is called static variable. It cannot be local. We can create a single copy of static variable and share it among all instances of the class.(memory allocation happens only once for static variables)

Operators:

Arithmetic operators:-

System.out.println(a+b);//15 System.out.println(a\*b);//50 System.out.println(a%b);//0

System.out.println(a/b);//2 System.out.println(a-b);//5

Unary operators:

1)Incrementing/decrementing. 2) negating an expression. 3)inverting the value of a Boolean.

System.out.println(x++);//10 (11)

System.out.println(++x);//12

System.out.println(x--);//12 (11)

System.out.println(--x);//10

System.out.println(~a);//-11 (minus of total positive value which starts from 0)

System.out.println(~b);//9 (positive of total minus, positive starts from 0)

System.out.println(!c);//false (opposite of boolean value)

System.out.println(!d);//true

TRANSIENT:-

The transient is used with the instance variable to eliminate it from the serialization process. During serialization, The value of transient field is not saved.

VOLATILE:-

The volatile is used to mark the JVM and thread to read its value from primary memory and not utilize cached value present in the thread stack. (It is used in concurrent programming in java).

NOTE:

Serialization is the concept of representing an object’s state as a byte stream. The byte stream has all info about object.

**JAVA CONTROL STATEMENTS | CONTROL FLOW IN JAVA: -**

1. **Decision making statement**

**. if statements**

**. switch statements**

1. **Loop statements**

**. do while loop**

**. while loop**

**. For loop**

**. For each loop**

1. **Jump statements**

**. break statement**

**. continue statement**

**->**Decision making statements decide which statements to execute and when

If statement:

1. Simple if statement.

if(condition) {

statement 1; //executes when condition is true

}

1. If-else statement:

if(condition) {

statement 1; //executes when condition is true

}

else{

statement 2; //executes when condition is false

}

1. If-else-if ladder:(if statement followed by multiple else- if statements).

if(condition 1) {

statement 1; //executes when condition 1 is true

}

else if(condition 2) {

statement 2; //executes when condition 2 is true

}

else {

statement 2; //executes when all the conditions are false

}

1. Nested if-statement:( if statement can contain a if or if-else statement inside another if or else-if statement)

if(condition 1) {

statement 1; //executes when condition 1 is true

if(condition 2) {

statement 2; //executes when condition 2 is true

}

else{

statement 2; //executes when condition 2 is false

}

**Switch statements:**

The case variable can be int, short,byte char or enumeration. String also supported since java 7 of java.

Cases cannot be duplicate.

Default statement is executed when any of the case doesn’t match the value of expression.

Break statement terminates the switch block when the condition is satisfied.(It is optional if not used next case is executed).

**public class Student implements Cloneable {**

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

**int num = 2;**

**switch (num){**

**case 0:**

**System.out.println("number is 0");**

**break;**

**case 1:**

**System.out.println("number is 1");**

**break;**

**default:**

**System.out.println(num);**

}}}

**LOOP STATEMENTS: (To execute block of code repeatedly)**

1)For loop 2) While loop 3) do-while loop

->we use for loop only when we exactly know the number of times, we want to execute the block of code.

Note:-

The enhanced version of for loop is for-each loop. To traverse the data structures like array or collection.In for each loop we don’t need to update the loop variable.

**for(data\_type var : array\_name/collection\_name){**

**//statements**

**} }**

->If we don’t know the exact number of iterations in advance, It is recommended to use a while loop.(It is also called as entry controlled loop. If the condition is true, then the loop body will be executed)

**public** **class** Calculation {

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

// TODO Auto-generated method stub

**int** i = 0;

System.out.println("Printing the list of first 10 even numbers \n");

**while**(i<=10) {

System.out.println(i);

i = i + 2;

}}}

->The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of executions is not known and we have to execute the loop at least once, we can use do-while loop.(it is also known as exit control loop since the condition is not checked in advance).

**Objects and classes in java:**

An entity that has state and behaviour is known as an object. It can be physical or logical(tangible or intangible). The example of intangible object is banking system. An object has 3 behaviours

1)state(represents the data of an object)

2)behaviour(represents the behaviour of an object such as deposit ,withdraw etc..,

3)Identity(It is used internally by the JVM to identify each object uniquely)

An object is an instance of a class: A class is a blue print or template from which objects are created. So, an object is the instance of a class.

CLASS:

A class is a group of objects which have common properties. It is a template or blue print from which objects are created. It is a logical entity. It can’t be physical.

A class contains fields, methods, constructors, blocks, nested class and interface.

Method (is usedto expose the behaviour of an object)

Advantages : code reusability and code optimization.

New keyword(is used to allocate memory at runtime).

**3 ways to initialize objects:**

1. By reference variable

  Student s1=**new** Student();

  s1.id=101;

  s1.name="Sonoo";

1. By Metod

**class** Student{

**int** rollno;

 String name;

**void** insertRecord(**int** r, String n){

   rollno=r;

   name=n;

  }

**void** displayInformation(){System.out.println(rollno+" "+name);}

}

**class** TestStudent4{

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

  Student s1=**new** Student();

  Student s2=**new** Student();

  s1.insertRecord(111,"Karan");

   s2.insertRecord(222,"Aryan");

  s1.displayInformation();

  s2.displayInformation();

 }}

**Anonymous object:**

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If we have to use an object only once, an anonymous object is a good approach.

Ex: calling method through reference

Calculator c = new Calculator():

c.fact(5);

Ex: calling method through anonymous object

new Calculator().fact(5);

**Constructor:**

A constructor is a block of code similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

**Note**: It is called constructor because it constructs the values at the values at the time of object creation. It is not necessary to create a constructor for a class. It is because java compiler creates a default constructor if your class doesn’t have any.

There is a constructor class in java. The purpose of this constructor class is to get internal information of a constructor in the class. It is found in the java.lang.reflect package.

**Rules for creating a constructor:**

**.** constructor name must be same as its class name.

. A constructor must have no explicit return type.

. A java constructor cannot be abstract, static, final and synchronized.

**Java Static Key word:**

The static key word in java is mainly used for memory management. We can apply static key word with variables, methods, blocks and nested classes.

->The static variable can be used to refer to the common property of all objects.

->The static variable gets memory only once in the class area at the time of class loading.

**Static method:**

->A static method belongs to the class rather than the object of a class.

->A static method can be invoked without the need for creating an instance of a class.

-> A static method can access static data member and can change the value of it.

**Static block:**

->Is used initialize static data member.

-> it is executed before the main method at the time of class loading.

**Questions:**

1)Can we execute a program without main()?

No, one of the ways was the static block, but it was possible till JDK 1.6. since JDK 1.7, it is not possible to execute a java class without main.

2)can we apply static key word with a top-level class?

No, static keyword cannot be applied with outer or top-level class but an inner class can be static.

3)can we use static variables with instance methods and instance variables with static methods?

Static means a class level accessibility.

Instance means an object level accessibility.

So we can call static variables from instance methods. But cannot call instance variables directly from static methods, because they have to be associated with an instance (object of a class).

If we want to access instance variable from static method. Create an object of that class and call this static variable.

public class Myclass

{

private int x = 10;

static int m1()

{

Myclass obj = new Myclass();

int y = obj.x;

return y;

}

public static void main(String[] args) { output:10

System.out.println(m1());

}}

4)Can we overload static methods?

The answer is yes. We can have two or more static methods with the same name, but differences in input parameters.

**public** **static** **void** foo() {

        System.out.println("Test.foo() called ");

    }

**public** **static** **void** foo(**int** a) {

        System.out.println("Test.foo(int) called ");

    }

5) can we override static methods?

We can declare static methods with the same signature in the subclass, but it is not considered overriding as there won’t be any runtime polymorphism. Hence the answer is ‘NO’(If a derived class defines a static method with the same signature as a static method in the base class, the method in the derived class is hidden by the method in the base class)

|  |
| --- |
| **class** Base {  **public** **static** **void** display() {          System.out.println("Static or class method from Base");      }  **public** **void** print()  {           System.out.println("Non-static or Instance method from Base");      }  }    // Subclass  **class** Derived **extends** Base {        // This method is hidden by display() in Base  **public** **static** **void** display() {           System.out.println("Static or class method from Derived");      }        // This method overrides print() in Base  **public** **void** print() {           System.out.println("Non-static or Instance method from Derived");     }  }    // Driver class  **public** **class** Test {  **public** **static** **void** main(String args[ ])  {         Base obj1 = **new** Derived();           // As per overriding rules this should call to class Derive's static         // overridden method. Since static method can not be overridden, it         // calls Base's display()         obj1.display();           // Here overriding works and Derive's print() is called         obj1.print();      }  } |

**Output**

Static or class method from Base

Non-static or Instance method from Derived

**this keyword in java:**

->this can be used to refer to current class instance variable.

->this can be used to invoke current class method(implicitly).

->**this()** can be used to invoke current class constructor.

->this can be passed as an argument in the method call.

->this can be passed as an argument in the constructor call.

->this can be used to return the current class instance from the method.

**Inheritence In java:(IS-A relationship)**

Inheritence in java is a mechanism in which one object acquires all the properties and behaviours of a parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When we inherit existing class, we can reuse methods and fields of the parent class. Moreover we can add new methods and fields in your current class also.

Why use inheritance in java?

* For method overriding( So runtime polymorphism can be achieved).
* For code reusability.

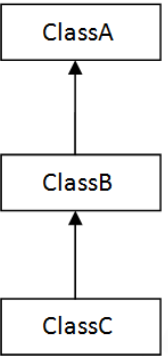
Types Of Inheritence:

**Single inheritance:**

When a class inherits another class, it is known as single inheritance.

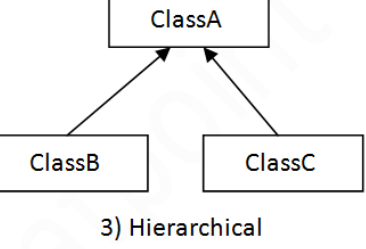
**Multilevel inheritance:**

When there is a chain of inheritance, it is known as multilevel inheritance.



**Hierarchial Inheritence:**

When two or more classes inherits a single class, It is known as hierarchial inheritance.



Note:

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A,B and C are 3 classes. The C class inherits A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

**Aggregation in java:**

If a class have an entity reference, it is known as aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many information such as id, name etc. It contains one more object named address, Which contains its own information such as city,state,country etc. In such case employee has an entity reference address, So relationship is employee HAS-A address.

When to use aggregation?

->Code reuses is also best achieved by aggregation when there is no is-a relationship.

->Inheritence should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

class Address{  
 String country;  
 String state;  
 String city;  
  
 public Address(String country, String state, String city) {  
 this.country = country;  
 this.state = state;  
 this.city = city;  
 }  
 }  
public class Emp{  
 int id;  
 String name;  
 Address address;  
 public Emp(int id, String name, Address address) {  
 this.id = id;  
 this.name = name;  
 this.address = address;  
 }  
 void display(){  
 System.*out*.println(id+""+name+""+address.country+""+address.state+""+address.city);  
 }  
  
 public static void main(String[] args) {  
 Address add1= new Address("India","AP","MPL");  
 Address add2= new Address("India","AP","MPL");  
  
 Emp e1= new Emp(1,"purushotham",add1);  
 Emp e2= new Emp(1,"purushotham",add2);  
 e1.display();  
 e2.display();  
 }}

**Method Overloading:**

If a class has multiple methods having same name but different in parameters, it is known as method overloading.

If we have to perform only one operation, having same name of the methods increase the readability of the program.(It increases the readability of the program).

Two ways to overload the method:

1. By changing number of arguments.
2. By changing the data type.

Note: In java, method overloading is not possible by changing the return type of the method only.

**Method overloading and Type promotion:**

The process of converting one type of object and variable into another type is known as type casting.

When the conversion automatically performs by the the compiler without the programmers interference, It is called implicit type casting or widening casting.

The process of converting lower data type to that of a higher datatype is referred to as widening.



In the same expression if we have two different types of variables are invoived:

1. The java compiler uses the predefined library function for transforming the variables.

2.The conversion of variable is done into a common datatype.

3.After that, expression is executed.

class OverloadingCalculation1{

  void sum(int a,long b){System.out.println(a+b);}

  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){

  OverloadingCalculation1 obj=new OverloadingCalculation1();

**obj.sum(20,20); //now second int literal will be promoted to long**

  obj.sum(20,20,20);

1. }}

Example of method overloading with type promotion in case of ambiguity?

class OverloadingCalculation3{

  void sum(int a,long b){System.out.println("a method invoked");}

  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){

  OverloadingCalculation3 obj=new OverloadingCalculation3();

  obj.sum(20,20);//now ambiguity

  }}

Output:Compile Time Error

**Method overriding:**

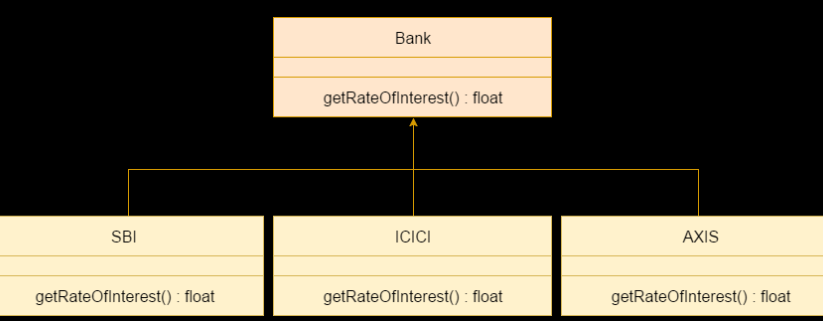
If subclass has the same method as declared in the parent class, it is known as method overriding in java. ->Method overriding is used to provide the specific implementation of a method which is already provided by its super class.

->Method overriding is used for runtime polymorphism.

Rules For method overriding:

1.The method must have the same name and same parameters as in the parent class

2.There must be an IS-A relationship(Inheritance).



**Access modifiers with method overriding:**

If we are overriding any method, overridden method(i.e. declared in subclass) must not be more restrictive.

class A{

protected void msg(){System.out.println("Hello java");}

}

public class Simple extends A{

void msg(){System.out.println("Hello java");} **//CompileTime Error**

 public static void main(String args[]){

 Simple obj=new Simple();

 obj.msg();

    }}

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

A method can have a different access modifier but it cannot be lower the access scope.

->Methods declared as public in a super class must be pubic in all subclasses.

->Methods declared protected in a super class must either be protected or public in subclass

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

**Diff between method overriding and overloading:**

|  |  |
| --- | --- |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

**Covariant return type:**

Before java 5.0, it was not possible to override a method by changing the return type. When we override a parent class method, the name, argument types and return type of the overriding method in child class has to be exactly same as that of parent class method.

Java 5.0 onwards it is possible to have diff return types for an overriding method in child class, but the child’s return type should be a subtype of parents return type.

Covariant return type works only for non-primitive return types.

Ex:

class SuperClass {

   SuperClass get() {

      System.out.println("SuperClass");

      return this;

   }}

public class Tester extends SuperClass {

   Tester get() {

      System.out.println("SubClass");

      return this;

   }

   public static void main(String[] args) {

      SuperClass tester = new Tester();

      tester.get();

   }}

OutPut:subclass

**Super Keyword in java**:

Super keyword in java is a reference variable which is used to refer immediate parent object.

Whenever we create a instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usages of super:

To refer immediate parent class instance variable.

To invoke immediate parent method.

Super() method can be used to invoke immediate parent class constructor.

**Runtime Polymorphism:**

Runtime polymorphismordynamic method dispatch is a processin which a call to an overridden method is resolved at runtime rather than compile time.

**ENCAPSULATION:**

Encapsulation in java is a process of wrapping code and data(variables)together into a single unit.

In, encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore it is also known as data hiding.

To achieve encapsulation:

->declare variables of a class as private

->provide public setter and getter methods to modify and view the variable values.

Advantages:

->The fields of a class can be made read-only or write-only.

->A class can have total control over what is stored in its fields..

**Absraction:**

Abstraction is the processs of hiding implementation details and showing only functionality to user.

Two ways to achieve abstraction in java

1.Abstract class(0-100%)

2.Interface(100%)

Abstract class:

A classs which is defined with abstract keyword is known as abstract class. It can have abstract and non-abstract methods. It needs to be extended and its methods implemented. It cannot be instantiated.

Abstraction let us focus on what the object does instead of how it does.

->class must be declared with abstract keyword.

->can have abstract and non-abstract methods

->It cannot be instantiated.

->It can have constructors and static methods also.

->It can have final methods which will force the subclass not to change the body of the method.

Example:

public abstract class BankAccount {

private String accountNumber;

private double balance;

// define Constructors, getters, and setters for accountNumber and balance

// Common methods for all types of accounts

public void deposit(double amount) {

// Implementation for deposit

} public void withdraw(double amount) {

// Implementation for withdrawal

} // Abstract method to be implemented by specific account types

public abstract void generateStatement();

}

// Concrete class representing a savings account

public class SavingsAccount extends BankAccount {

// Additional properties and methods specific to savings accounts

@Override

public void generateStatement() {

// Implementation for generating a savings account statement

}

}

// Concrete class representing a checking account

public class CheckingAccount extends BankAccount {

// Additional properties and methods specific to checking accounts

@Override

public void generateStatement() {

// Implementation for generating a checking account statement

}

}

The above bank account class is an abstract class. This means that it cannot be instantiated on its own. The calss has private fields “account number and balance” that represents common properties shared by all types of accounts. It has common methods that all types of accounts share such as “deposit and withdraw”. These methods provide a generic interface for interacting with any account regardless of its specific type. And it is having an abstract method   
“generate statement” and must be implemented by concrete subclasses.

This method ensures that each specific type of account must provide its own implementation for generating a statement.

By utilizing this structure, the banking application achieves abstraction in the following ways.

**Common interface**: All account types share a common interface through the “Bank account” aabstract class, providing a consistent way to interact with any account.

**Code resusability**: common functionalities, like deposit and withdrawa are implemented in one place(the bank account class), promoting code reusability.

**Enforced specific implementations:** Each specific account type must implement its unique functionality, such as generating a statement, ensuring that appropriate behaviour is defined for each type.

Note:

Static methods in abstact class frequently used to implement utility functions or other class level features.

**Utility methods**: static methods may offer common utility operations for all subclasses. For instance, any subclass that extends super class can utilize static methods of super class. As the same method need not be implemented in every subclass, this can decrease code duplication and save time.

**Functionality at the class level:** Static methods in abstract classes can offer functionality at the class level independent of any instance-specific attributes. Instead of being implemented independently in each concrete subclass, can use static method implemented in abstract class.

Abstract classes can have final methods which cannot be overridden.

**Interface in java:**

Interface in java is used to achieve abstraction and multiple inheritance. It can have abstract methods and variables.

Since java8 we can have default and static methods in an interface.

Since java 9 we can have private methods in an interface.

**Note:**

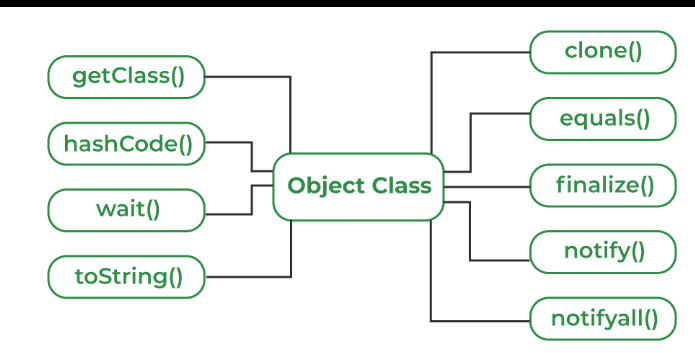
Private methods are meant to be used as helper methods within the interface itself and it cannot be overriden or accessed by implementing classes.

Private methods can be used to encapsulate common functionality that is shared among multiple default or static methods within the interface.

**Object class in java:**

The object class is the parent class of all the classes in java by default. In other words, it is the topmost class in java. If a class does not extend any other class then it is a direct child class of Object and if extend another class then it is indirectly derived.

So the object class methods will be available to all the classes.



1.toString():

The toString() provides a string representation of an object and is used to convert an object to a string. The default toString method for class object returns a string consisting of the name of the class of which the object is an instance, the at-sign character ‘@’, and the unsigned hexa decimal representation of hash code of the object.

2.hashcode():

For every object, JVM generates a unique number which is a hashcode. It returns distinct integer for distinct objects. It converts the internal address of object to an integer by using an algorithm.

The hash code method is native because in java it is impossible to find the address of an object, So it uses native languages like c/c++ to find the address of object.

**Use of hashcode()method:**

It returns a hash value that is used to search objects in a collection. JVM uses the hashcode method while saving objects into hashing-related data structures like HashSet,HashMap,HashTable,etc. The main advantage saving objects based on hash code is that searching becomes easy.

3.Clone()method:

It returns a new object that is exactly the same as this object.

The Java.lang.cloneable interface must be implemented by the class whose object clone we want create.If we don’t implement cloneable interface, clone method generates Clone Not Supported Exception.

The clone method saves the extra processing task for creating the exact copy of an object.If we perform it by using new keyword, it will take a lot of processing time to be performed.

4.getClass()

It returns the Class class object of this object. The Class class can further be used to get the metadata of this class.

5.equals():

It compares the given object to this object.

**Java Math Class:**

Java math class provides some emthods to work on math calculations.

Math.abs():It will return value of the given value.

Math.max(): will return the largest of two numbers.

Math.min():will return the smallest of two numbers.

Math.round: It is used to round of the decimal numbers to the nearest value.

Math.sqrt():It is used to return the squrt of a number.

Math.cbrt():It is used to return the cuberoot of a number.

Math.pow():It returns the value of first argument raised to the power of second argument.

 //returns 28 power of 4 i.e. 28\*28\*28\*28

 System.out.println("Power of x and y is: " + Math.pow(28, 4));

Math.random:It returns a double value with a positive sign,greater than or equal to 0.0 and less than 1.0.

**Wrapperclasses:**

The wrapper classes in java provides a mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into object is known autoboxing and vice-versa.

USE OF WRAPPER CLASSES:

Java is an object-oriented programming language, So we need to deal with objects many times like in collections, Serialization, Synchronization, etc.(Java collection framework works with objects only.All classes of the collection framework(ArrayList,Linkedlist,Vector.HashSet,LinkedHashset,Tree set,Priorityqueue,Dequeue etc.)deal with objects only.

**CommandLine Arguments:**

Java command line arguments is an argument i.e passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

**class** CommandLineExample{

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

System.out.println("Your first argument is: "+args[0]);

}}

**compile by** > javac CommandLineExample.java

**run by** > java CommandLineExample Purushotham

**Output: Your first argument is: Purushotham**

**Java String:**

In java, string is basically an object that represents sequence of char values. An array characters works same as java String.

The java.lang.String class implements Serializable,comparable and CharSequence interfaces.

->Char sequence interface is used to represent the sequence of characters. String,StringBuffer and String Builder classes implement it. We can create Strings in java by using these 3 classes.

How to create a string?

There are two ways to create a string 1. By string literal 2. By new Keyword.

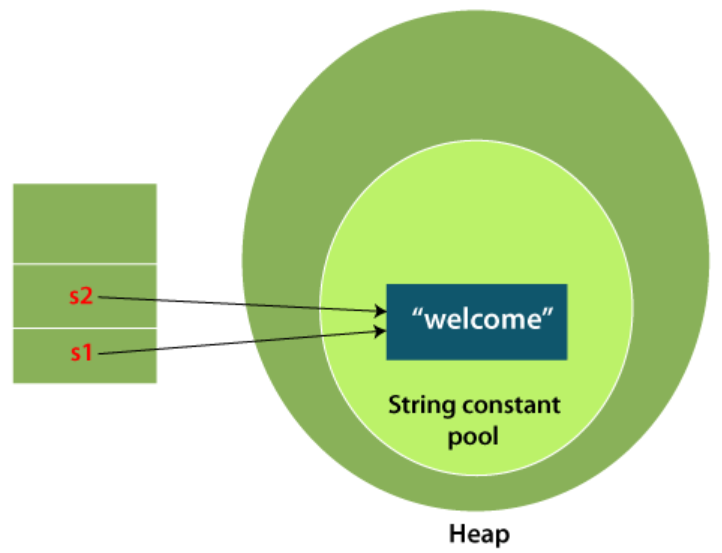
->String literal is created by using double quotes.

Ex: String s=”welcome”;

Each time when we create a string literal, JVM checks the “String constant pool” first. If the string already exists in the pool, JVM will return the reference to the same instance. If the string doesn’t exist in the pool, a new string instance is created and placed in the pool.

Ex: String s1=”Welcome”

String s2=”Welcome” // It does not create a new instance



**Note:String objects are stored in special memory called” String constant pool”.**

Why java uses the concept of string literal?

To make java more memory efficient (because no new objects are created if it exists already in string constant pool).

2)By new Keyword:

EX: String s= new String(“Welcome”);

In such case JVM will create a new String object in normal (non pool) heap memory, and the literal “Welcome” will be placed in the String constant pool. The variable s will refer to the object in heap(non-pool)

**Immutable String in java**:

Once the object is created its data or state can’t be changed but a new String object is created.

Ex:

Class TestImmutableString{

Public static void main(String args[]){

String s=”Sachin”;

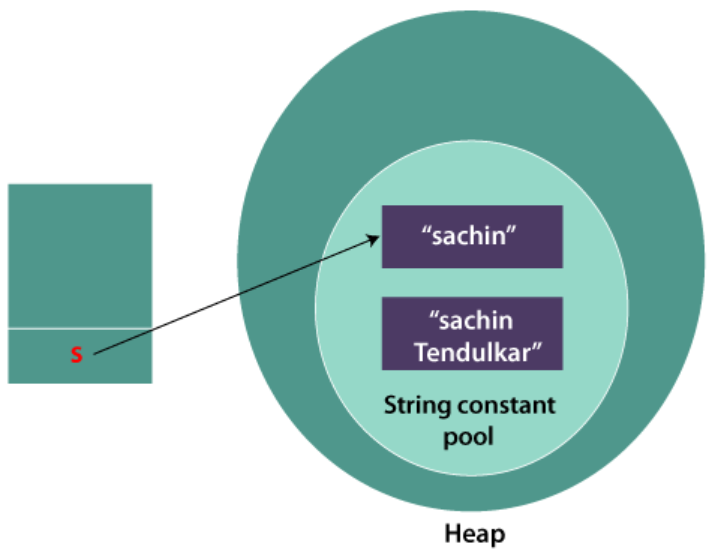
s.concat(“Tendulkar”); //appends the string at the end

System.out.println(s);

}}// OUTPUT:: Sachin

Note:

Here sachin is not changed but a new object is created with Sachin Tendulkar. That is y string is immutable.



As we can see 2 objects are created but s reference variable still refers to “Sachin” not to “Sachin tendulkar”. But if we explicitly assign to the reference, it will refer to “Sachin Tendulkar” object.

Class TestImmutableString{

Public static void main(String args[]){

String s=”Sachin”;

S=s.concat(“Tendulkar”);

System.out.println(s);

}}// OUTPUT:: Sachin Tendulkar

**Why string objects are immutable in jav?**

As java uses the concept of string literal. Suppose there are 5 reference variables, all refer to one object “Sachin”. If one reference variable changes the value of the object, it will be affected by all the reference variables. That is y string objects are immutable.

**String Concatination:**

In java, String concatenation forms a new string that is combination of multiple strings. 2 ways to concatenate strings.

1. By +(string concatenation)operator.
2. By concat() method.

Ex: String s= “Sachin”+”Tendulkar”;

Output: Sachin Tendulkar

Ex2: String s= 50+30+”Sachin”+40+40;

Output: 80Sachin4040

Note: after a string literal all the + will be treated as String concatenation operator.

2)By concat() method:

The string concat() method concatenates specified string to the end of the current string.

Ex: String s1=”Sachin”;

String s2=”Tendulkar”;

String s3=s1.concat(s2);

**String concatenation using StringBuilder class:**

String builder class provides append() method to perform concatenation. The append methos accepts arguments of different types like objects, Stringbuilder, int char, charsequence, Boolean, float, double.

Ex: StringBuilder s1= new StringBuilder(“Hello”);

StringBuilder s2= new StringBuilder(“world”);

StringBuilder s=s1.append(s2);

**Using format() method:**

String format method allows to concatenate multiple strings using format specifier like %s followed by string values or objects.

Ex: String s1= new String(“Hello”);

String s2= new String(“world”);

String s= String.format(“%s%s”,s1,s2)

System.out.println(s.toString);

**Using String Join():**

String join method is available from java8 and above versions. String.join() method accepts arguments first a separator and an array of string objects.

Ex:String s1= new String(“Hello”);

String s2= new String(“world”);

String s= String.join(“”,s1,s2);

System.out.println(s.toString());

**Using String Joiner class:**

String joiner class has all the functionalities of string.join(). In advance its constructor can also accept optional arguments.

EX: StringJoiner s= new StringJoiner(“ ,”);

s.add(“hello”);

s.add(“world”);

**Using collectors.joining():**

The collectors class in java 8 offers joining() method that concatenates the input elements in a similar order as they occur.

Ex: List<String> liststr= Arrays.asList(“abc”,”pqr”,”xyz”);

String str=liststr.stream().collect(collectors.joining(“ ,”));

System.out.println(str.toString());

**Substring in java:**

A part of string is substring. String class provides built-in substring() method that extract a substring from the given string using the index values passed. Incase of substring method startindex is inclusive and end index in exclusive.

**public String substring(int startIndex):**

String str="Hello";  
System.*out*.println(str.substring(2)); //output:llo

**public String substring(int startIndex, int endIndex):**

String s="hello”;

System.out.println(s.substring(0,2)); // output: he

**Using String.split() method**:

The split() method of string class can be used to extract a substring from a sentence.

Ex: String str="Hello, My name is purushotham";  
String[] arr=str.split("\\.");  
System.*out*.println(Arrays.*toString*(arr));

OutPut: [Hello, My name is purushotham]

**StringMethods:**

toUppercase() method converts string into uppercasr and toLowercase() converts string into lower case.

Ex: String s= “Sachin”;

System.out.println(s.toUpperCase());

System.out.println(s.toLowerCase());

**String trim():**

String trim() method eliminates white spaces before and after the string

Ex:String s=” Sachin “;// sachin

System.out.println(s.trim());//sachin

**String startsWith() and endsWith():**

The startsWith() checks whether the string starts with the letters passed as an argument and endsWith() methos checks whether the string ends with the letters passed as an argument.

Ex:String s=”Sachin”;

System.out.println(s.startsWith(“sa”));//true

System.out.println(s.endsWith (“n”));//true

**String charAt():**

The charAt() method returns a character at specified index.

Ex:String s=”Sachin”;

System.out.println(s.charAt(0));

**String length():**

Returns the length of specified string.

**String valueOf():**

The string class valueOf() methods converts given type such as int, long, float, double, boolean, char and char array into string.

Ex:int a=10;

String s= String.valueOf(a);

System.out.println(s+10)//1010

**String replace():**

replace() method replaces the first sequence of characters with second sequence of characters.

String s=”java is a programming language and java ia platform independent”);

String replace=s.replace(“java”,”kava”);//replaces all occurrences of “java” to “kava”

**Java StringBuffer class:**

Java StringBuffer class is used to create mutable string objects. i.e it can be changed.

**Note:** Java Stringbuffer class is thread-safe i.e multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Constructors of string buffer class:

StringBuffer(): It creates an empty string buffer with the initial capacity of 16.

String Buffer(string str): It creates a string buffer with the specified string.

StringBuffer(int capacity): It creates an empty string buffer with the specified capacity as length.

**Methods of stringbuffer:**

**String Buffer append():**

It concatenates given argument with this string. Ex: StringBUffer sb= new StringBuffer(“Hello”);

System.out.println( Sb.append(“java”));// Hello java

**String Buffer insert():**

It inserts the given string with this string at the given position. Ex: StringBuffer sb= new StringBUffer(“Hello”);

System.out.println( Sb.insert(1,”java”));//Hjavaello

**StringBUffer replace():**

The replace method replaces the given string from the specified beginindex and endindex.

Ex: StringBuffer sb=new StringBuffer(“Hello”);

System.out.println(sb.replace(1,3,java));// HJavalo

**StringBUffer delete():**

Delete() methods deletes the string from specified beginindex to end index. Ex: StringBuffer sb= new StringBUffer(“Helllo”);

System.out.println(sb.delete(1,3));// Hlo

**StringBuffer reverse():**

StringBUffer sb= new StringBUffer(“Hello”);

System.out.println(sb.reverse());// olleH

**Java StringBuilder class:**

Java string builder class is used to create mutable string. It is same a sstringbuffer class except that it is non-synchronised.

Constructors:

StringBuilder(): It creates an empty stringbuilder with initial capacity of 16.

StringBUilder(String str): It creates a string builder with specified string.

StringBUilder(int length): It creates an empty string builder with the specified capacity as length.

Methods:

public StringBuilder append(String s)

public StringBuilder insert(int offset, String s)

public StringBuilder replace(int startIndex, int endIndex, String str)

public StringBuilder delete(int startIndex, int endIndex)

public StringBuilder reverse()

public char charAt(int index)

|  |  |
| --- | --- |
| public String substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

**Multi threading:**

**What is process and thread?**

Thread is a smallest unit of execution within a process or program.

Many threads can run concurrently within a program.

At run time, threads in a program exist in a common memory space and can, therefore, share both data and code(i.e., they are light weight compared to processes).

Note: The programs which we were running so far, all those programs are single threaded. Why because java has its thread called main thread.

Process is an independent program in execution.

**Multi tasking:**

Multi tasking allows several activities to occur concurrently on the computer.

1. Process-based multi tasking(multiple programs or processes running parallelly)
2. Thread-based multi tasking(multiple threads run within a program)

Process-based multi tasking allows processes (i.e programs) to run concurrently on the computer.

Eg: Running the Ms paint while also working with the word processor.

Thread-based multi tasking allows parts of the same program to run concurrently on the computer.

Eg: MS word that is printing and formatting text at the same time and suggesting spellings of the word is not correct.

**Thread vs Process:**

Two threads share the same address space.

Context switching between threads is usually less expensive than between processes.

The cost of communication between threads is relatively low.

**Why multi-threading?**

In a single threaded environment, only one task at a time can be performed. So CPU cycles are wasted, for example, when waiting for user input.

Multi tasking allows idle CPU time to be put to good use.

**THE MAIN THREAD:**

When a standalone application runs, a user thread is automatically created to execute the main() method of the application. This thread is called main thread.

If no other user threads are spawned, the program terminates when the main() method finishes executing.

All other threads, called child threads, are spawned from the main thread.

The main() method can then finish, but the program will keep running until all user threads have completed.

The runtime environment distinguishes between user threads and daemon threads.

**Note:**

If there is no user threads running program will stop irrespective of whether the daemon threads are running or not.

Calling the setDaemon(Boolean) method in the thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.

As long as user threads is alive, the JVM does not terminate.

Daemon threads are referred to as a low priority threads,these are required for supporting background tasks like garbage collection, releasing memory of unused objects etc.,

**How to create our own thread:**

A thread in java is represented by an object of the thread class.

Creating threads is achieved in one of two ways.

1. Implementing the java.lang.Runnable interface.

->create a userdefined class and make that class extends runnable interface.

->override run method.

->instantiate an object(create object) of a class that implements the runnable interface.

->create a Thread object and pass our runnable object to its constructor.

public class MyRunnable implements Runnable {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getId() + " Value " + i);

}

}

public static void main(String[] args) {

MyRunnable myRunnable = new MyRunnable();

Thread thread1 = new Thread(myRunnable);

Thread thread2 = new Thread(myRunnable);

thread1.start();

thread2.start();

}

1. Extending the java.lang.Thread class.

->Create an userdefined class and extends with thread class

->override run() method

->create instance of userdefined class.

->start the thread.

public class MyThread extends Thread {

@Override

public void run() {

for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getId() + " Value " + i);

}

}

public static void main(String[] args) {

MyThread thread1 = new MyThread();

MyThread thread2 = new MyThread();

thread1.start();

thread2.start();

}}

**Synchronization in java:**

Threads share same memory space, i.e. they can share resources(Objects).

Note: Threads can share resources, so it allows multiple threads to access same resource.

However, there are some critical situations where it is desirable that only one thread at a time has access to a shared resource. So here synchronization allows only one thread to access the resorce.

While a thread is inside a synchronized method of an object, all other threads that wish to execute this method will have to wait.

**The Volatile Keyword:**

In java, the ‘volatile’ keyword is used to indicate that a variables value may be changed by multiple threads simultaneously. The primary purpose of ‘volatile’ keyword is to prevent threads from caching the variables value locally.

When a variable declared as volatile, All reads and writes of the volatile are directly done from and to main memory.

**Producer consumer problem:**

Producer-consumer problem also known as bounded-buffer problem is a classic example of a multiprocess synchronization problem**.**

The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue

* The producers job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is removing data from the buffer.

**Problem:**

To make sure that the producer wont try to add data into the buffer if its full and that the consumer wont try to remove data from an empty buffer.

Solution:

The producer is to either go to sleep if the buffer is full. The next time the consumer removes an item from the buffer,It notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

**public** **class** Threadexample {

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

**throws** InterruptedException

    {

        // Object of a class that has both produce()

        // and consume() methods

**final** PC pc = **new** PC();

        // Create producer thread

        Thread t1 = **new** Thread(**new** Runnable() {

            @Override

**public** **void** run()

            {

**try** {

                    pc.produce();

                }

**catch** (InterruptedException e) {

                    e.printStackTrace();

                }}

        });

        // Create consumer thread

        Thread t2 = **new** Thread(**new** Runnable() {

            @Override

**public** **void** run()

            {

**try** {

                    pc.consume();

                }

**catch** (InterruptedException e) {

                    e.printStackTrace();

                }}

        });

   t1.start();

        t2.start();

        // t1 finishes before t2

        t1.join();

        t2.join();}

**public** **static** **class** PC {

        // Create a list shared by producer and consumer

        // Size of list is 2.

        LinkedList<Integer> list = **new** LinkedList<>();

**int** capacity = 2;

        // Function called by producer thread

**public** **void** produce() **throws** InterruptedException

        {

**int** value = 0;

**while** (**true**) {

**synchronized** (**this**)

                {

                    // producer thread waits while list

                    // is full

**while** (list.size() == capacity)

                        wait();

                    System.out.println("Producer produced-"

                                       + value);

                    // to insert the jobs in the list

                    list.add(value++);

                    // notifies the consumer thread that

                    // now it can start consuming

                    notify();

                    // makes the working of program easier

                    // to  understand

                    Thread.sleep(1000);

                }}}

        // Function called by consumer thread

**public** **void** consume() **throws** InterruptedException

        {

**while** (**true**) {

**synchronized** (**this**)

                {

                    // consumer thread waits while list

                    // is empty

**while** (list.size() == 0)

                        wait();

                    // to retrieve the first job in the list

**int** val = list.removeFirst();

                    System.out.println("Consumer consumed-"

                                     + val);

                    // Wake up producer thread

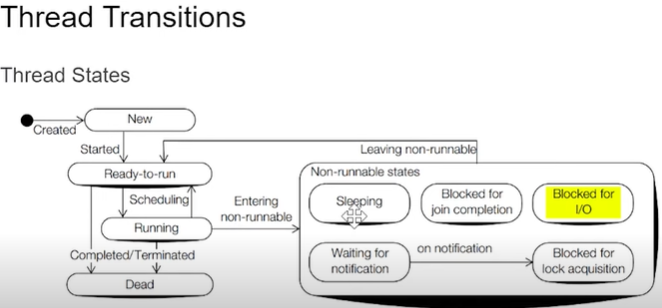
                    notify();

                    // and sleep

                    Thread.sleep(1000);

                }}}}}

**Lifecycle of a thread:**



Note: whenever a thread is notified or awakened it does not directly jump into the runnable state , it means it will go to “blocked for lock acquisition state” and fight for lock then it can start execution.

A close-up of a text

Description automatically generated

Wait(): this method causes the current thread to wait and go to sleep until some other threads call the notify() or notifyll() method for the objects monitor(lock). It simply releases the lock and mostly used for inter-thread communication. It should only be called from a synchronized context.

notify(): wakes up only a single thread instead of multiple threads that are waiting on the objects monitor.

notifyAll(): wakes up all the threads.

Join() method is generally used to pause the execution of a current thread until another thread finishes its execution.

Notify:

Invoking the notify() method on an object wakesup a single thread that is waiting for the lock of this object.

On being notified, a waiting thread first transists to the blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready to run state.

**Thread priorities** :

Thread are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled.

Priorities are integer values from 1 (lowest priority given by the constant Thread.MIN\_PRIORITY) to 10(highest priority given by the constant Thread.MAX\_PRIORITY). The default priority is 5(Thread.NORM\_PRIORITY).

The priority of a thread can be set using the” setPriority()” and read using “getPriority()”.

**Thread scheduler:**

Scheduler in JVM implementations usually employ one of the two following strategies.

1. Preemptive scheduling.

If a thread has a higher priority than current running thread, It moves to the current running thread to ready to run state to execute this thread.

1. Time sliced or round-robin scheduling.

Where a running thread is allowed to run for a fixed length of time, after that it moves to ready-to-run state and waits for its turn to run again.

**DeadLock:**

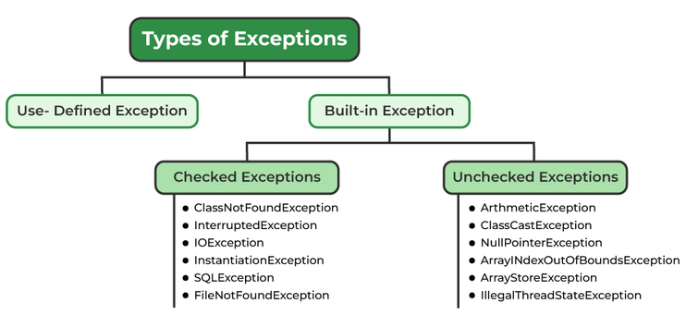
**Exception handling:**

**In java exception is unexpected event that occurs during execution of a program, i.e is at runtime that disrupts normal flow of the program’s instructions.**

**Error:**

**Errors represents irrecoverable condition such as JVM running out of memory, memory leaks, stack overflow errors, infinite recursion etc.,**

**Types of exceptions:**

****

**Exceptions can be categorized into two types**

1. **Built in exception**
2. **User-defined exception**

**Built in exceptions are the exceptions that available in java libraries.**

**Checked exceptions:**

**Checked exceptions are called compile time exceptions because these exceptions are checked at compile-time by the compiler.**

**Unchecked exceptions:**

**Unchecked exceptions are just opposite to checked exceptions. The compiler will not check these exceptions at compile time.**

**User defined exceptions:**

**Sometimes, the built in exceptions in java are not able to describe a situation. In such cases, users can also create exceptions,**