**Java OOPs Concepts**

**Simula** is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

**Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

**Object** means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

**Object**

Any **entity** that has **state** and **behaviour** is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

**Class**

**Collection of objects** is called class. It is a logical entity.

**Inheritance**

**When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

**Polymorphism**

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convenes the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meow, dog barks woof etc.

**Abstraction**

**Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.

**Encapsulation**

**Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

**Advantage of OOPs over Procedure-oriented programming language**

|  |
| --- |
| 1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows. |
| 2)OOPs provides data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere. |
| 3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.  **3 R**’s :- **Readablility**, **Reusability** and **Reliability** |

|  |  |
| --- | --- |
|  |  |

**What is difference between object-oriented programming language and object-based programming language?**

|  |
| --- |
| Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages. |

Advantage of naming conventions in java

By using standard Java naming conventions, we make our code easier to read for ourselves and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

|  |  |
| --- | --- |
| **Name** | **Convention** |
| class name | should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc. |
| interface name | should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc. |
| method name | should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc. |
| variable name | should start with lowercase letter e.g. firstName, orderNumber etc. |
| package name | should be in lowercase letter e.g. java, lang, sql, util etc. |
| constants name | should be in uppercase letter. e.g. RED, YELLOW, MAX\_PRIORITY etc. |

**Object and Class in Java**

**Object in Java**

An entity that has state and behaviour is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

* **state:** represents data (value) of an object.
* **behaviour:** represents the behaviour (functionality) of an object such as deposit, withdraw etc.
* **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

|  |
| --- |
| For Example: Pen is an object. Its name is Reynolds, colour is white etc. known as its state. It is used to write, so writing is its behaviour. |

|  |
| --- |
| **Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance(result) of a class. |

**Class in Java**

|  |
| --- |
| A class is a group of objects that has common properties. It is a template or blueprint from which objects are created. |

A class in java can contain:

* **data member**
* **method**
* **constructor**
* **block**
* **class and interface**

**Instance variable in Java**

|  |
| --- |
| A variable that is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when object(instance) is created. That is why, it is known as instance variable. |

**Method in Java**

|  |
| --- |
| In java, a method is like function i.e. used to expose behaviour of an object. |

**Advantage of Method**

* Code Reusability
* Code Optimization

**new keyword**

|  |
| --- |
| The new keyword is used to allocate memory at runtime. |
|  |

object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area.

There are many ways to create an object in java. They are:

* **By new keyword**
* **By newInstance() method**
* **By clone() method**
* **By factory method etc. getInstance()**

**Annonymous object**

|  |
| --- |
| **A**nnonymous simply means nameless. An object that have no reference is known as annonymous object. |
| If we have to use an object only once, annonymous object is a good approach. |

**new** Calculation().fact(5);//calling method with annonymous object

Creating multiple objects by one type only

|  |
| --- |
| We can create multiple objects by one type only as we do in case of primitives. |

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();//creating two objects

# Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

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

## Advantage of method overloading?

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

|  |
| --- |
| Different ways to overload the method  There are two ways to overload the method in java |

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

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

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity and produces compile time error as :-Method is already defined

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

Yes, by method overloading. We can have any number of main methods in a class by method overloading.

Method Overloading and Type Promotion

**One type is promoted to another implicitly if no matching data-type is found**

**Example of Method Overloading with Type Promotion**

**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);

  }

}

Output:40

60

### Example of Method Overloading with Type Promotion if matching found

**If there are matching type arguments in the method, type promotion is not performed.**

**class** OverloadingCalculation2{

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

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

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

  OverloadingCalculation2 obj=**new** OverloadingCalculation2();

  obj.sum(20,20);//now int arg sum() method gets invoked

  }

}

Output: int arg method invoked

### Example of Method Overloading with Type Promotion in case ambiguity

**If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be 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

|  |  |
| --- | --- |
| * **why java main method is static?**   Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.  **Java static block**   * Is used to initialize the static data member. * It is executed before main method at the time of classloading. * Can we execute a program without main() method?   Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.  JDK 1.7 :- Output:Error: Main method not found in class A3, please define the main method as:  public static void main(String[] args)  **Java static nested class**  A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.   * It can access static data members of outer class including private. * Static nested class cannot access non-static (instance) data member or method.  this keyword in java There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object. Usage of java this keyword  1. **this keyword can be used to refer current class instance variable.**   If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.   1. **this() can be used to invoke current class constructor.**   The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if we have many constructors in the class and want to reuse that constructor.  ***Call to this() must be the first statement in constructor.***   1. **this keyword can be used to invoke current class method (implicitly)**   We may invoke the method of the current class by using the this keyword. If we don't use the this keyword, compiler automatically adds this keyword while invoking the method.   1. **this can be passed as an argument in the method call.**   The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.   1. **this can be passed as argument in the constructor call.**   Same as point 4   1. **this keyword can also be used to return the current class instance.**   We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).  java this keyword Inheritance in Java (Is-A) **Inheritance in java** is a mechanism in which one object acquires all the properties and behaviours of parent object.  The idea behind inheritance in java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of parent class, and we can add new methods and fields also.  Inheritance represents the **IS-A relationship**, also known as parent-child relationship. Why use inheritance in java  * For Method Overriding (so runtime polymorphism can be achieved). * For Code Reusability.  Syntax of Java Inheritance  1. **class** Subclass-name **extends** Superclass-name 2. { 3. //methods and fields 4. }   **Why multiple inheritance is not supported in java?**  To reduce the complexity and simplify the language, multiple inheritance is not supported in java.  Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and we call it from child class object, there will be ambiguity to call method of A or B class.  Since compile time errors are better than runtime errors, java renders compile time error if we inherit 2 classes. So whether we have same method or different, there will be compile time error now.  C:\Users\PIYUSH\Desktop\Inheritance in Java   Javatpoint.png  **Aggregation in Java (Has-A)**  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, email-Id etc. It contains one more object named address, which contains its own information such as city, state, country, zipcode etc. as given below.   1. **class** Employee{ 2. **int** id; 3. String name; 4. Address address;//Address is a class 5. ... 6. }   In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.  **When use Aggregation?**   * Code reuse is also best achieved by aggregation when there is no **is-a** relationship. * Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.  Method Overriding in Java If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.  In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding. Usage of Java Method Overriding  * Method overriding is used to provide specific implementation of a method that is already provided by its super class. * Method overriding is used for runtime polymorphism  Rules for Java Method Overriding  1. method must have same name as in the parent class 2. method must have same parameter as in the parent class. 3. must be **IS-A** relationship (inheritance).   **Can we override static method?**  No, static method cannot be overridden. It can be proved by runtime polymorphism.  **Why we cannot override static method?**  because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.  **Can we override java main method?**  No, because main is a static method.  **Java access modifiers with method overriding**  If we are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.   1. **class** A{ 2. **protected** **void** msg(){System.out.println("Hello java");} 3. } 5. **public** **class** Simple **extends** A{ 6. **void** msg(){System.out.println("Hello java");}//C.T.Error 7. **public** **static** **void** main(String args[]){ 8. Simple obj=**new** Simple(); 9. obj.msg(); 10. } 11. }  |  | | --- | | **The default modifier is more restrictive than protected. That is why there is compile time error.** |   **Exception Handling with Method Overriding in Java** |
| * + **If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.**   + **If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.** |

**Difference between constructor and method in java**

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if we don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

**Rules for creating java constructor**

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

**There are two types of constructors:**

1. Default constructor (no-arg constructor)
2. Parameterized constructor

**Java Copy Constructor**

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

* By constructor

Student6 s1 = **new** Student6(111,"Karan");

Student6 s2 = **new** Student6(s1);

* By assigning the values of one object into another

Student7 s1 = **new** Student7(111,"Karan");

Student7 s2 = **new** Student7();

s2.id=s1.id;

     s2.name=s1.name;

* By clone() method of Object class

class should implement Cloneable interface

Student7 s1 = new Student7(111,"Karan");

Student7 s3 = ((Student7 )s1.clone()); // this should either be caught or the declaring method should throw CloneNotSupportedException

**Object Cloning in Java**

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

**protected** Object clone() **throws** **CloneNotSupportedException**

**Why use clone() method ?**

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

**Java static keyword**

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

**Java static variable**

If we declare any variable as static, it is known static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
* The static variable gets memory only once in class area at the time of class loading.

**static** **int** count=0; //will get memory only once and retain its value

**int** count=0; //will get memory when instance is created

**Java static method**

If we apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* static method can access static data member and can change the value of it.

**Restrictions for static method**

|  |
| --- |
| There are two main restrictions for the static method. They are: |

|  |
| --- |
| 1. The static method cannot use non static data member or call non-static method directly. 2. this and super cannot be used in static context. |

**Two Public classes in same JAVA file :-**

Not allowed :-

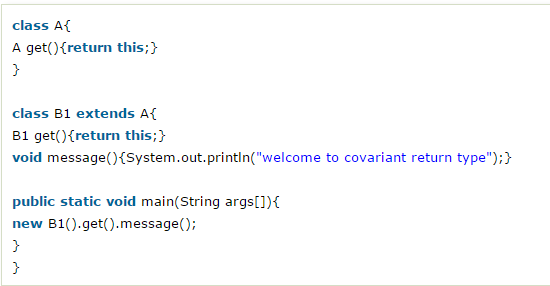
class SampleClass is public, should be declared in a file named SampleClass.java

**Covariant Return Type**

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type

Different Return type for same overridden method.



**super keyword in java**

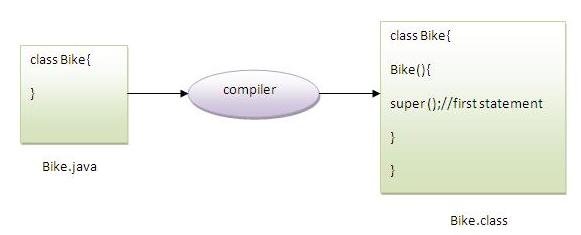
The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

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

**Usage of java super Keyword**

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.

***super() is added in each class constructor automatically by compiler.***



1. super is used to invoke immediate parent class method.

# Instance initialize block

|  |
| --- |
| **Instance Initialize block** is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initialize block. |

**Rules for instance initializer block :**

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.

**The java compiler copies the code of instance initializer block in every constructor.**

1. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
2. The instance initializer block comes in the order in which they appear.

# Final Keyword In Java

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

1. variable
2. method
3. class

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

1. **Java final variable**

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

1. **Java final method**

If we make any method as final, we cannot override it though we can call the final method in child class.

1. **Java final class**

If we make any class as final, we cannot extend it.

**Blank or uninitialized final variable :-**

* A final variable that is not initialized at the time of declaration is known as blank final variable.
* A final variable can be initialized only in Constructor.

**Static blank final variable :-**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

**final parameter**

If we declare any parameter as final, we cannot change the value of it.

**Polymorphism in Java**

**Polymorphism in java** is a concept by which we can perform a *single action by 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.

If we overload static method in java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

**Runtime Polymorphism in Java**

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Let's first understand the upcasting before Runtime Polymorphism.

class Bike {

public void speed() {}

}

class Honda extends Bike {

public void speed() { System.out.println(“Speed of Bike”);}

public static void main(String args[]) {

Bike b = new Honda(); // upcasting

b.speed();

}

}

Output: **Speed of Bike**

**Java Runtime Polymorphism with data member**

|  |
| --- |
| Method is overridden not the data members, so runtime polymorphism can't be achieved by data members. |

**class** Bike{

**int** speedlimit=90;

}

**class** Honda3 **extends** Bike{

**int** speedlimit=150;

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

  Bike obj=**new** Honda3();

  System.out.println(obj.speedlimit);//90

}

=🡺 Output:90

# Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

### Understanding Type

Let's understand the type of instance.

#### 1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

**int** data=30;

Here data variable is a type of int.

#### 2) References have a type

**class** Dog{

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

  Dog d1;//Here d1 is a type of Dog

 }

}

#### 3) Objects have a type

|  |
| --- |
| An object is an instance of particular java class, but it is also an instance of its superclass. |

**class** Animal{}

**class** Dog **extends** Animal{

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

  Dog d1=**new** Dog();

 }

}

|  |
| --- |
| Here d1 is an instance of Dog class, but it is also an instance of Animal. |

### static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

### Example of static binding

**class** Dog {

**private** **void** eat(){System.out.println("dog is eating...");}

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

  Dog d1=**new** Dog();

 d1.eat();

}

}

Output : dog is eating...

### Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

### Example of dynamic binding

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){

System.out.println("dog is eating...");

}

}

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

  Animal a=**new** Dog();

  a.eat();

}

}

Output: dog is eating...

**What’s the point of having a private constructor?**

* The private modifier when applied to a constructor works in much the same way as when applied to a normal method or even an instance variable. Defining a constructor with the private modifier says that only the native class (as in the class in which the private constructor is defined) is allowed to create an instance of the class, and no other caller is permitted to do so.
* To implement a singleton. A singleton is a design pattern that allows only one instance of one class to be created, and this can be accomplished by using a private constructor.
* To prevent object construction entirely. When would it make sense to do something like that? Of course, when creating an object doesn’t make sense – and this occurs when the class only contains static members. And when a class contains only static members, those members can be accessed using only the class name – no instance of the class needs to be created.
* Runtime, Desktop, java.awt.Toolkit are example of Singleton classes

**What is an abstraction ?**

Ans) Abstraction is a way of converting real world objects in terms of class. It’s a concept of defining an idea in terms of classes or interface. For example creating a class Vehicle and injecting properties into it. E.g

public class Vehicle {

public String colour;

public String model;

}

**What is Encapsulation?**

Ans) The encapsulation is achieved by combining the methods and attribute into a class. The class acts like a container encapsulating the properties. The users are exposed mainly public methods. The idea behind is to hide how things work and just exposing the requests a user can do.

The **Java Bean** class is the example of fully encapsulated class.

By providing only setter or getter method, we can make the class **read-only or write-only**.

**What is Association?**

Ans) Association is a relationship where all object have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

**What is Aggregation?**

Ans) Aggregation is a specialize form of Association **where all object have their own lifecycle but there is ownership and child object cannot belongs to another parent object**. Let's take an example of Department and teacher. A single teacher cannot belongs to multiple departments, but if we delete the department teacher object will not destroy. We can think about "has-a" relationship.

Example :- Student has Address. Student is an object and Address is an object. Address object cannot belong to other object. Student A having Address A gets shifted to some other Address then Address A can be used by some other Student.

**What is Composition ?**

Ans) Composition is again specialize form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. **Child object does not have their lifecycle** and **if parent object is deleted all child object will also be deleted**. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room cannot belongs to two different house if we delete the house room will automatically delete.

**What is marker or tagged interface?**

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

**public** **interface** Serializable{

}

**Two basic design purposes of tagging interfaces:**

**Creates a common parent**: As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

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

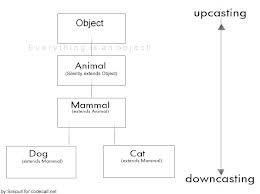
# Java instanceof

The java **instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false.

**Important** :- If we apply the instanceof operator with any variable that has null value, it returns false

**Upcasting and downcasting**



is Cat an Object - It doesn't extend Object, it extends Mammal?  
By inheritance Cat gets all the properties its ancestors have. Object is Cat's grandgrandparent, which means Cat is also an Object. Cat is also an Animal and a Mammal, which logically means - if Mammals possess mammary glands and Animals are living beings, then Cat also has mammary glands and is living being.

**by casting we are not actually changing the object itself, we are just labelling it differently.**

For example, if we create a Cat and upcast it to Animal, then the object doesn't stop from being a Cat. It's still a Cat, but it's just treated as any other Animal and it's Cat properties are hidden until it's downcasted to a Cat again.  
Let's look at object's code before and after upcasting:

Cat c = new Cat();

System.out.println(c);

Mammal m = c; // upcasting

System.out.println(m);

/\*

This printed:

Cat@a90653

Cat@a90653

\*/

As we can see, Cat is still exactly the same Cat after upcasting, it didn't change to a Mammal, it's just being labelled Mammal right now. This is allowed, because Cat is a Mammal.

Although there's no need to for programmer to upcast manually, it's allowed to do.  
Consider the following example:

Mammal m = (Mammal)new Cat();

is equal to

Mammal m = new Cat();

But downcasting must always be done manually:

Cat c1 = new Cat();

Animal a = c1; //automatic upcasting to Animal

Cat c2 = (Cat) a; //manual downcasting back to a Cat

* **Why is that so, that upcasting is automatically, but downcasting must be manual?**

Well, we see, upcasting can never fail. But if we have a group of different Animals and want to downcast them all to a Cat, then there's a chance, that some of these Animals are actually Dogs, and process fails, by throwing ClassCastException.

**Double buffering, movement, and collision detection**

This is where is should introduce an useful feature called "instanceof", which tests if an object is instance of some Class.  
Consider the following example:

Cat c1 = new Cat();

Animal a = c1; //upcasting to Animal

if(a instanceof Cat){ // testing if the Animal is a Cat

System.out.println("It's a Cat! Now I can safely downcast it to a Cat, without a fear of failure.");

Cat c2 = (Cat)a;

}

Note, that casting can't always be done in both ways. If we are creating a Mammal, by calling "new Mammal()", we a creating a Object that is a Mammal, but it cannot be downcasted to Dog or Cat, because it's neither of them.  
For example:

Mammal m = new Mammal();

Cat c = (Cat)m;

Such code passes compiling, but throws "java.lang.ClassCastException: Mammal cannot be cast to Cat" exception during running, because we are trying to cast a Mammal, which is not a Cat, to a Cat.  
  
General idea behind casting, is that, which object is which. We should ask, is Cat a Mammal? Yes, it is - that means, it can be cast.  
Is Mammal a Cat? No it isn't - it cannot be cast.  
Is Cat a Dog? No, it cannot be cast.  
Important: Do not confuse variables with instances here. Cat from Mammal Variable can be cast to a Cat, but Mammal from Mammal variable cannot be cast to a Cat.

**Abstraction in Java**

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

**Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

**Abstract Class**

A class which contains the abstract keyword in its declaration is known as abstract class.

* Abstract classes may or may not contain abstract methods ie., methods without body ( public void get(); )
* But, if a class have at least one abstract method, then the class must be declared abstract.
* If a class is declared abstract it cannot be instantiated.
* To use an abstract class we have to inherit it from another class, provide implementations to the abstract methods in it.
* If we inherit an abstract class we have to provide implementations to all the abstract methods in it.

**Abstract Methods**

If we want a class to contain a particular method but we want the actual implementation of that method to be determined by child classes, we can declare the method in the parent class as abstract.

* abstract keyword is used to declare the method as abstract.
* We have to place the abstract keyword before the method name in the method declaration.
* An abstract method contains a method signature, but no method body.
* Instead of curly braces an abstract method will have a semicolon ( ; ) at the end.

Declaring a method as abstract has two consequences:

* The class containing it must be declared as abstract.
* Any class inheriting the current class must either override the abstract method or declare itself as abstract.

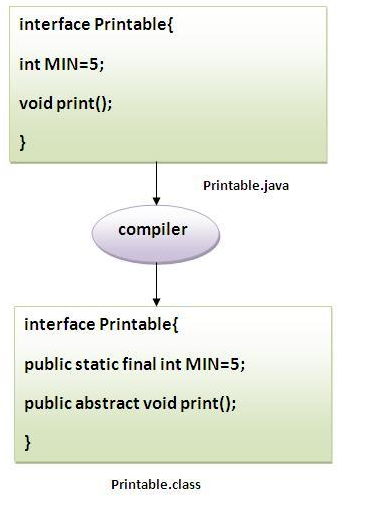
**Interfaces**

***The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.***

* We cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.
* An implementation class itself can be abstract and if so interface methods need not be implemented.
* Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

1. Interface classes can have public methods, default methods and static methods
2. Data members are public, static and final be default
3. Declaring methods or data members as private will give below error :- **modifier private not allowed here**
4. Non-Abstract methods in Interfaces can be added via static methods which does not needs to be overridden in child class and the same can be added via **InterfaceName.staticMethodName()** or by using inner class;
5. Defining non-abstract methods without static in interface will give below error :-

**interface abstract methods cannot have body**

****

**Concrete methods in the inner class of an Interface**

package com.manish.innerclass;

interface Printable {

void print();

class Printer {

public void getName() {

System.out.println("Printer Name is :- Epson");

}

}

}

public class NonAbstractInInterface implements Printable {

public void print() {

System.out.println("Printing in Test Class");

}

public static void main(String args[]) {

NonAbstractInInterface naip = new NonAbstractInInterface();

naip.print();

**NonAbstractInInterface.Printer printer = new NonAbstraceInInterface.Printer();**

printer.getName();

}

}

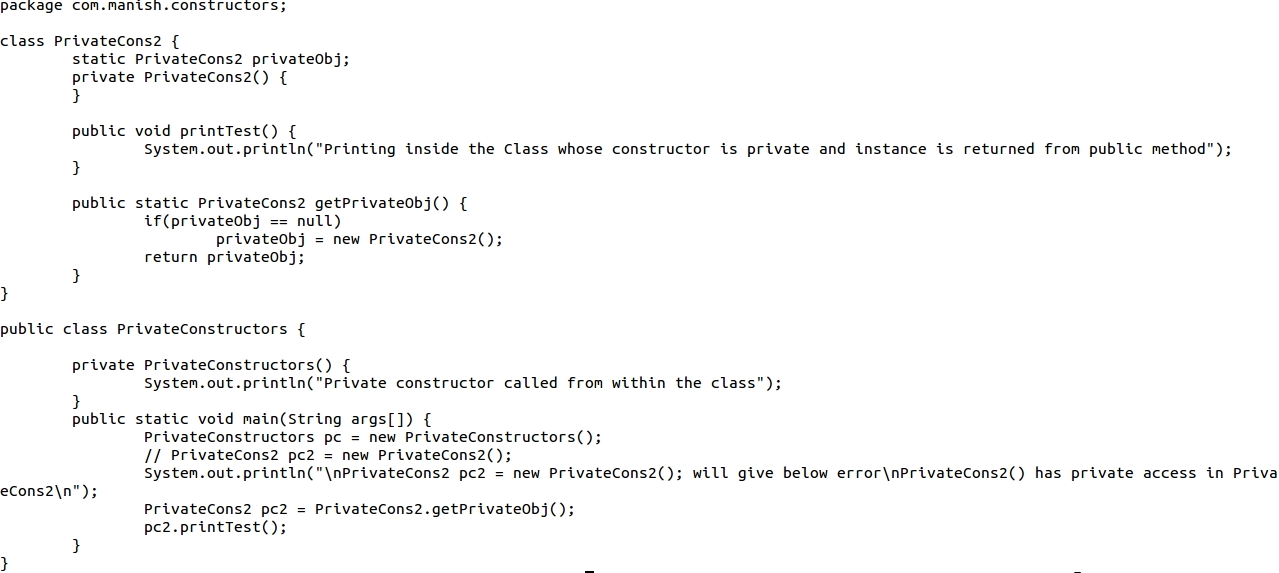
**Modifiers:-**

C:\Users\PIYUSH\Downloads\Untitled Diagram.png

**Access Modifiers**

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

* If a constructor is made private then it prevents creation of instance of the class from outside of the class and are used to create Singleton classes.



* **Protected access modifier can only be applied to methods and data members and cannot be applied to class** **because**:-

Protected class member (method or variable) is just like package-private (default visibility), except that it also can be accessed from subclasses.  
Since there's no such concept as '**subpackage**' or **'package-inheritance**' in Java, declaring class protected or package-private would be the same thing.

**Non Access Modifiers**

Java provides a number of non-access modifiers to achieve many other functionality.

* The *static* modifier for creating class methods and variables
* The *final* modifier for finalizing the implementations of classes, methods, and variables.
* The *abstract* modifier for creating abstract classes and methods.
* The *synchronized* and *volatile* modifiers, which are used for threads
  1. **static modifier**
     + **static variables** :- Only one copy of the static variable exists regardless of the number of instances of the class. Static variables are also known as class variables. Local variables cannot be declared static.
     + **static methods** :-  Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
  2. **final modifier**
     + **final variables** - final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object. *final* modifier often is used with *static* to make the constant a class variable.
     + **final methods** - A final method cannot be overridden by any subclasses. The main intention of making a method final would be that the content of the method should not be changed by any outsider.
     + **final class** - The main purpose of using a class being declared as *final* is to prevent the class from being sub-classed.
  3. **abstract modifier**
     + **abstract class** - An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.
     + **abstract methods** - An abstract method is a method declared without any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.
  4. **synchronized modifier**

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

* 1. **transient modifier**

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

public transient int limit = 55; // will not persist

public int b; // will persist

* 1. **volatile modifier**
     + The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.
     + Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

public class MyRunnable implements Runnable{

private volatile boolean active;

public void run(){

active = true;

while (active){ // line 1

// some code here

}

}

public void stop(){

active = false; // line 2

}

}

* + - Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

**Java Strictfp Keyword**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform.

* The strictfp keyword can be applied on methods, classes and interfaces.

**strictfp** **class** A{}//strictfp applied on class

**strictfp** **interface** M{}//strictfp applied on interface

**class** A{

**strictfp** **void** m(){}//strictfp applied on method

}

* The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

**class** B{

**strictfp** **abstract** **void** m();//Illegal combination of modifiers

}

**class** B{

**strictfp** **int** data=10;//modifier strictfp not allowed here

}

**class** B{

**strictfp** B(){}//modifier strictfp not allowed here

}

|  |  |
| --- | --- |
| **Object** | **Class** |
| Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. |
| Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| Object is a **physical** entity. | Class is a **logical** entity. |
| Object is created through **new keyword** mainly e.g. Student s1=new Student(); | Class is declared using **class keyword** e.g. class Student{} |
| Object is created **many times** as per requirement. | Class is declared **once**. |
| Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| 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. |

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. |
| Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| Abstract class can be inherited by a class or an abstract class | Interfaces can be extended only by interfaces. Classes has to implement them instead of extend |
| In abstract class, the keyword ‘abstract’ is mandatory to declare a method as an abstract | In interfaces, the keyword ‘abstract’ is optional to declare a method as an abstract because all the methods are abstract by default |
| Abstract class can have protected , public and public abstract methods | Interface can have only public abstract methods i.e. by default |
| Abstract class **can have main method and constructor**. | Interface **can't have main method or constructor**. |
| Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |

Whether to use an interface or abstract class.

**Interface:**

1. To **define a contract** ( preferably stateless - I mean no variables )
2. To link unrelated classes with "**has-a**" capabilities.
3. To declare public constant variables (**immutable state**)

**Abstract class:**

1. Share code among several closely related classes. It establishes "**is-a**" relation.
2. Share common state among **related classes** ( state can be modified in concrete classes)

**Unrelated classes can have capabilities through interface but related classes change the behaviour through extension of base classes.**