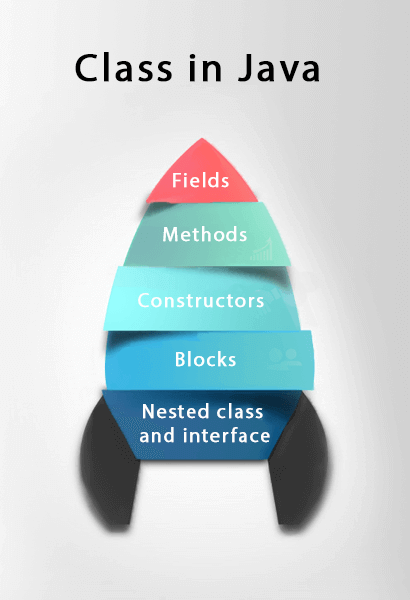
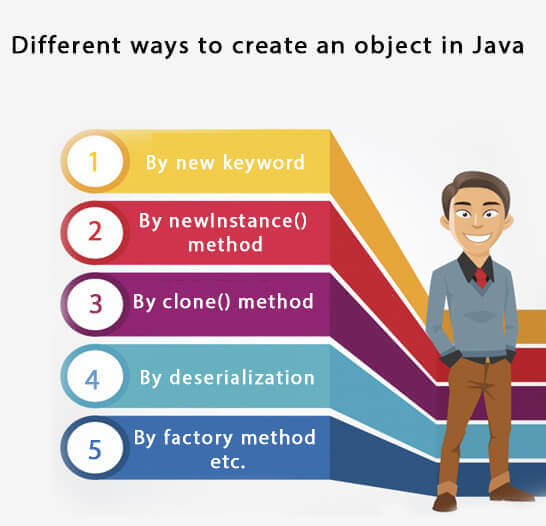
OOP



Class – blueprint



Object – instance of a class that has identity, state and behavior, Object is the root class of all classes in Java







**Variable types – primitive vs reference**

**Instance variables** are created inside the class but outside the method. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created.

**Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. Invoked at the time of object creation

**class** Bike7{

**int** speed;

    Bike7(){System.out.println("speed is "+speed);}

    {speed=100;}

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

    Bike7 b1=**new** Bike7();  // prints speed is 100

    Bike7 b2=**new** Bike7();  // prints speed is 100

    }

}

**Static** keyword belongs to the class than an instance of the class

**Final** keyword (variable can NOT be changed, class can NOT be extended and method can NOT be overridden), blank final variable can only be initialized in constructor

**class** Bike10{

**final** **int** speedlimit;//blank final variable

  Bike10(){

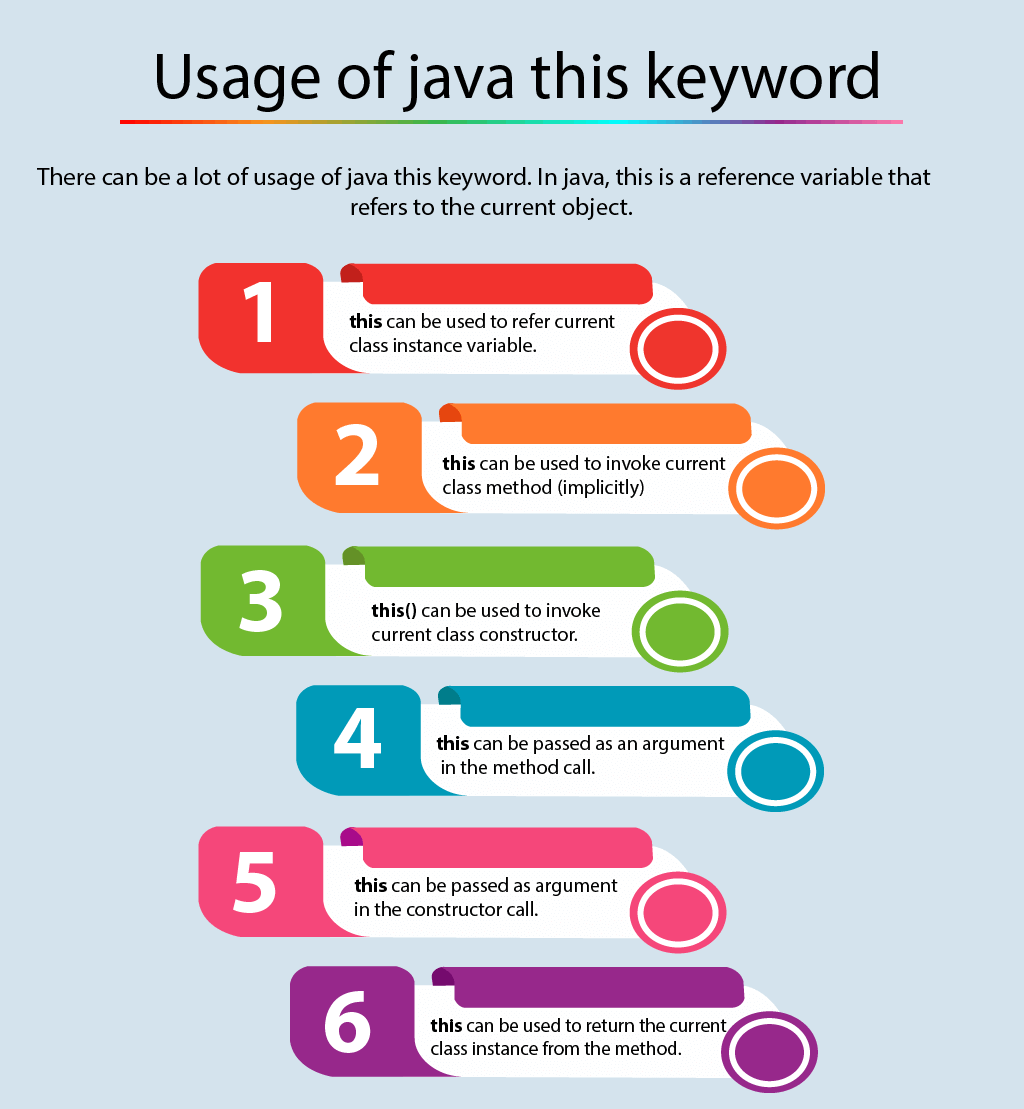
  speedlimit=70;

  System.out.println(speedlimit);

  }

}

**this** is a **reference variable** that refers to the current object



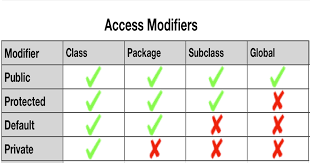
Java does NOT support multiple inheritance (ie: class C extends A, B where A and B have the same method – compile time error. Use multiple interfaces instead)

**Encapsulation** – binding code/data into single unit accessible through getters/setters, modifiers

Packages – built in and user defined

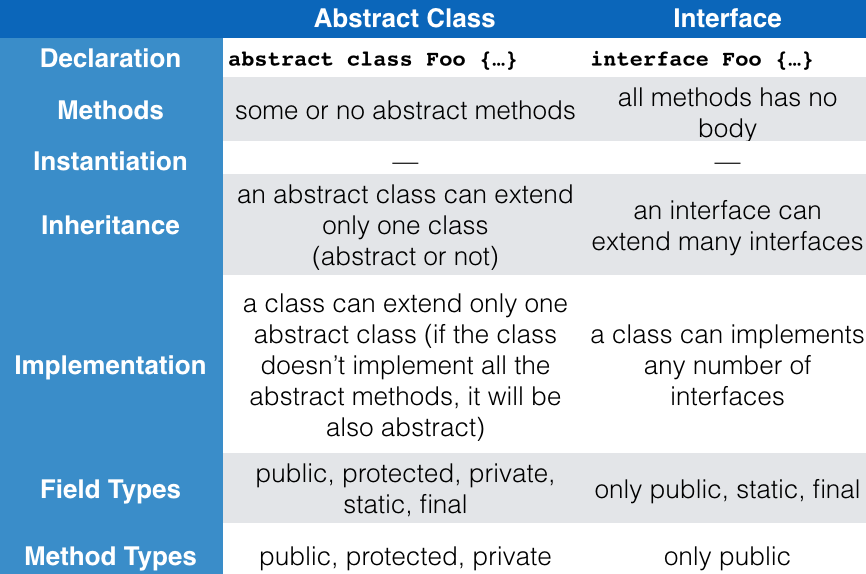
Modifiers- access and non

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.



**Abstraction** – hiding implementation and showing functionality





Reasons to choose Interface



**abstract** **class** Bank{

**abstract** **int** getRateOfInterest();

}

**class** SBI **extends** Bank{

**int** getRateOfInterest(){**return** 7;}

}

**class** PNB **extends** Bank{

**int** getRateOfInterest(){**return** 8;}

}

**abstract** **class** Bike{

   Bike(){System.out.println("bike is created");}

**abstract** **void** run();

**void** changeGear(){System.out.println("gear changed");}

 }

//Creating a Child class which inherits Abstract class

**class** Honda **extends** Bike{

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

 }

//Creating a Test class which calls abstract and non-abstract methods

**class** TestAbstraction2{

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

  Bike obj = **new** Honda();

  obj.run();

  obj.changeGear();

 }

}

**interface** Printable{

**void** print();

}

**interface** Showable{

**void** show();

}

**class** A7 **implements** Printable,Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **void** show(){System.out.println("Welcome");}

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

A7 obj = **new** A7();

obj.print();

obj.show();

 }

}

**Polymorphism** – one task performed in different ways (method overriding and overloading), runtime and compiletime

**Method overloading** – change number of arguments OR change data types

**Method overriding** – subclass with same method as parent class (can’t override static method)

Covariant return type – method overriding where return type is different. Below example A’s get returns A while B’s get returns B

**class** A{

A get(){**return** **this**;}

}

**class** B1 **extends** A{

B1 get(){**return** **this**;}

**void** message(){System.out.println("welcome to covariant return type");}

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

**new** B1().get().message();

}

}

**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.

A method is overridden, **not the data members**, so runtime polymorphism can't be achieved by data members.

**Upcasting** - reference variable of Parent class refers to the object of Child class

**class** Bike{

**int** speedlimit=90;

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

}

**class** Splendor **extends** Bike{

**int** speedlimit=150;

**void** run(){System.out.println("running safely with 60km");}

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

    Bike b = **new** Splendor(); //upcasting

    b.run();  // prints running safely with 60km

Bike a = new Bike();

a.run(); // prints running

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

  }

}

**class** Animal{

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

}

**class** Dog **extends** Animal{

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

}

**class** BabyDog1 **extends** Dog{

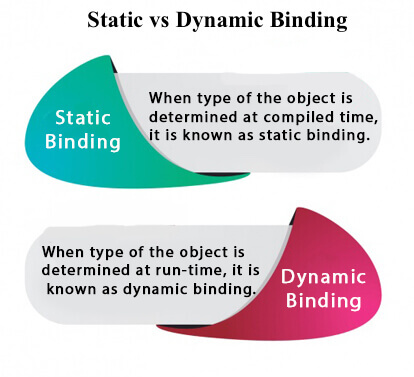
**public** **static** **void** main(String args[]){

Animal a=**new** BabyDog1();

a.eat();

}}

// prints dog is eating bc BabyDog is not overriding the eat() method, so eat() method of Dog class is invoked



If there is any private, final or static method in a class, there is 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();

 }

}

In the below example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.

**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();

 }

}

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

**class** Simple1{

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

 Simple1 s=**new** Simple1();

 System.out.println(s **instanceof** Simple1);//true

 }

}

**interface** Printable{}

**class** A **implements** Printable{

**public** **void** a(){System.out.println("a method");}

}

**class** B **implements** Printable{

**public** **void** b(){System.out.println("b method");}

}

**class** Call{

**void** invoke(Printable p){//upcasting

**if**(p **instanceof** A){

A a=(A)p;//Downcasting

a.a();

}

**if**(p **instanceof** B){

B b=(B)p;//Downcasting

b.b();

}

}

}//end of Call class

**class** Test4{

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

Printable p=**new** B();

Call c=**new** Call();

c.invoke(p);  // outputs ‘b method’

}

}

**Inheritance** – when one object acquires all of the properties and behaviors of its parent object

**Coupling** - Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

**Cohesion** - Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

**Association** - Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* **One to One**
* **One to Many**
* **Many to One**
* **Many to Many**

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be undirectional or bidirectional.

**Aggregation** - Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a ***has-a*** relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

**class** Employee{

**int** id;

Address address;//Address is a class

...

}

**Composition** - The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

*Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

**(Auto)boxing** (value to obj) vs **Unboxing**

Integer number = 100; // autoboxing

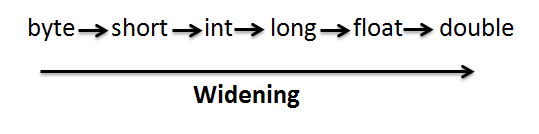
int inum = number; // unboxing

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

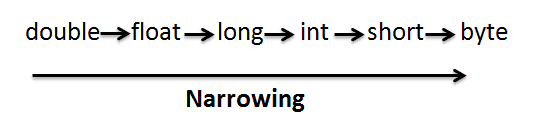
Casting: **Implicit (widen)** vs **Explicit (narrow)** Conversion

**Type Casting** in Java is  nothing but converting a primitive, interface or class into another type.

Implicit – safe, no risk of losing data



Explicit – it is likely conversion could lose data



<https://javainterviewpoint.com/type-casting-java-implicit-explicit-casting/>

An ***enum*** *type* is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.

C# ONLY - struct