Object Oriented Programming in Java

Er.Sital Prasad Mandal

BCA- 2nd sem
Mechi Campus
Bhadrapur, Jhapa, Nepal
(Email : info.sitalmandal@gmail.com)
https://ctaljava.blogspot.com/

Text Book

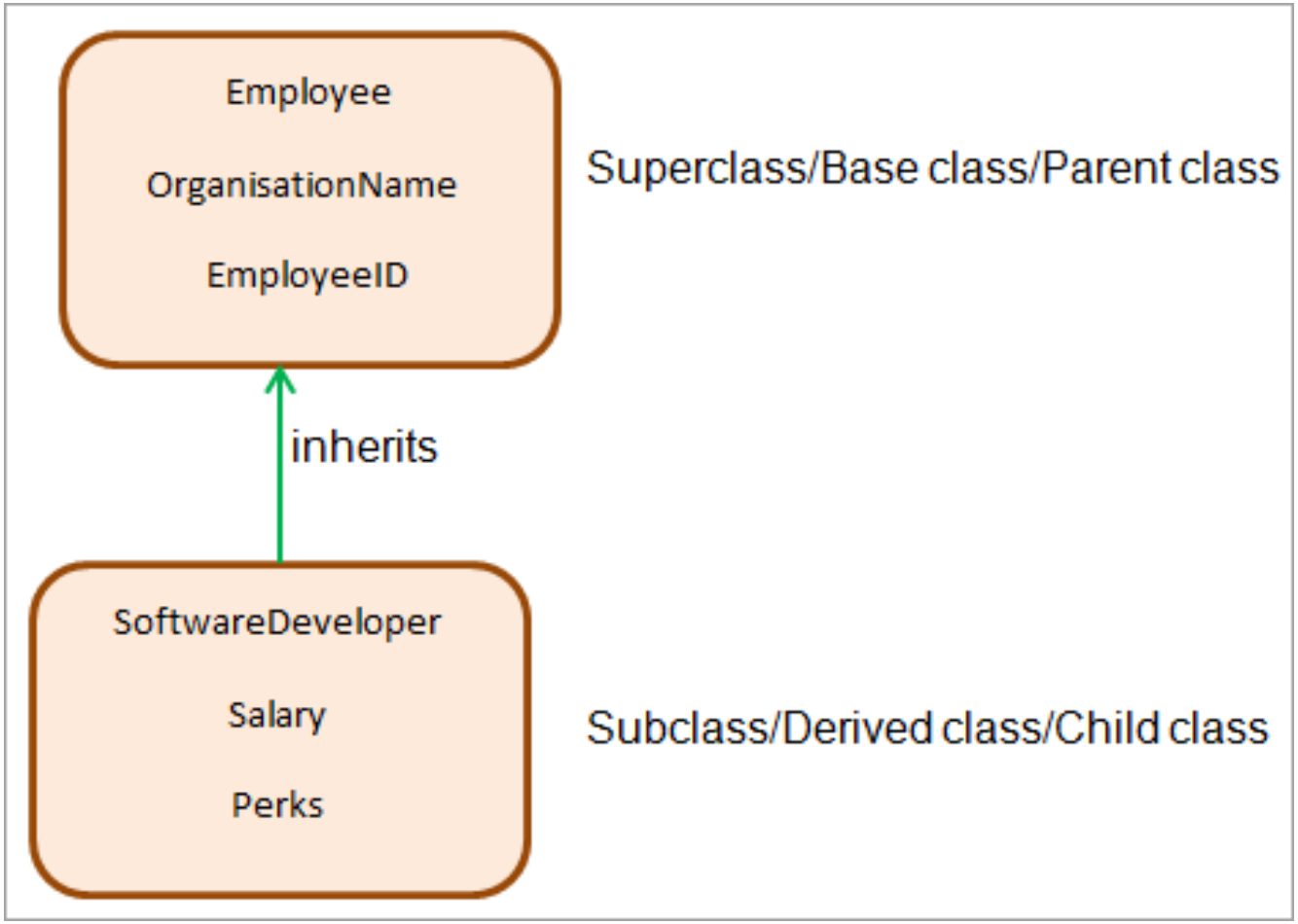
1. Deitel & Dietel. -Java: How to-program-. 9th Edition. TearsorrEducation. 2011, ISBN: 9780273759168

2. Herbert Schildt. "Java: The CoriviaeReferi4.ic e 61 Seventh Edition. McGraw -Hill 2006, 1SBN; 0072263857

Inheritance & Packaging

1. Inheritance:
   - ✓ Using 'extends' keyword
   - ✓ Subclasses and Superclasses
   - ✓ 'super' keyword usage
2. Overriding Methods, Dynamic Method Dispatch
3. The Object class, Abstract and Final Classes
4. Package: Access Control
5. Interfaces:
   - ✓ Defining an Interface
   - ✓ Implementing and applying interfaces.

Inheritance



Inheritance

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**.

The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.

Child Class:
The class that extends the features of another class is known as child class, sub class or derived class.

Parent Class:
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

*Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.*

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.

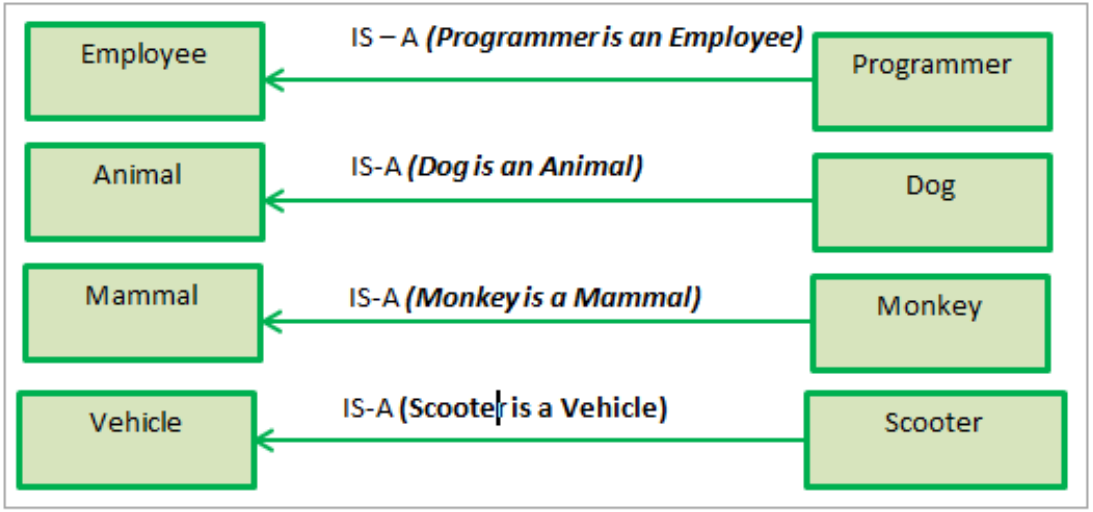
Inheritance allows us to reuse of code, it improves reusability in your java application.

Inheritance

Syntax: Inheritance in Java

To inherit a class we use **extends** keyword. Here class `Mammal` is child class and class Animal is parent class. The class `Mammal` is inheriting the properties and methods of Animal class.

```
class Mammal extends Animal {
}
```



Inheritance (IS-A)

When we talk about inheritance, the most commonly used keyword would be **extends and implements (interface later on).**

These words would determine whether one object **IS-A type** of another. By using these keywords we can make one object acquire the properties of another object.

IS-A Relationship:

IS-A is a way of saying : This object is a type of that object.

Let us see how the **extends keyword is used to achieve inheritance.**

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

Inheritance

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well.

Inheritance

```
//Parent class or Superclass or base class
class Superclass {
   // private doesn't takes
    int num = 100;
}
//Child class or subclass or derived class
class Subclass extends Superclass {
    /* The same variable num is declared in the Subclass
     * which is already present in the Superclass
     */
    int num = 110;
    public static void main ( String args[] ) {
        Subclass obj = new Subclass();
        obj.printNumber();                          //Output: 110
        Superclass so = new Superclass();
        System.out.println(so.num );
    }
    void printNumber () {
        System.out.println(num);
    }
}
```

With use of the **extends** keyword the subclasses will be able to inherit all the properties of the superclass **except** for the **private** properties of the superclass.

Inheritance

The instanceof Keyword:

Let us use the **instanceof operator** to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

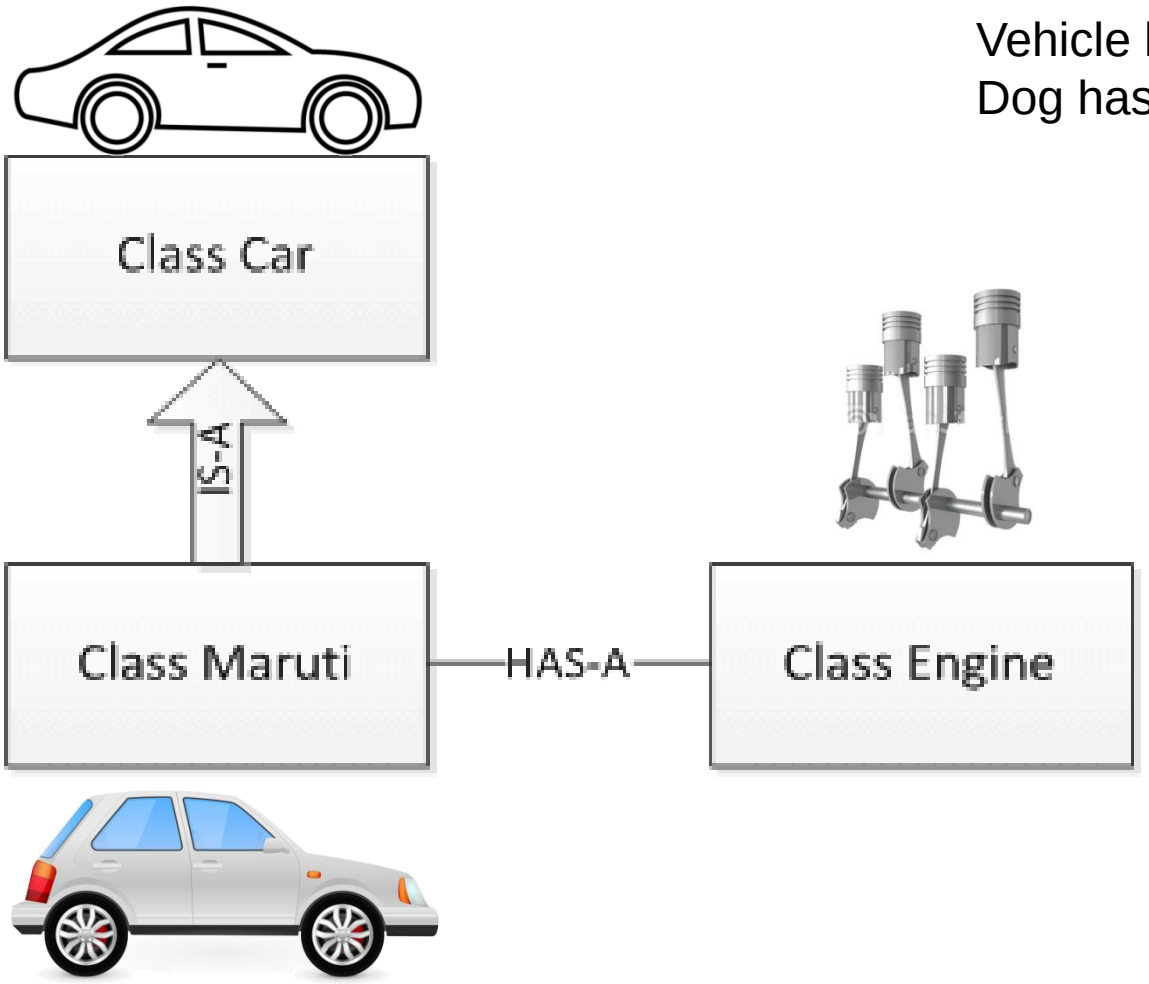
```
class Animal {
}
class Mammal extends Animal {
}
public class Dog extends Mammal {
    public static void main ( String args[] ) {
        //  Animal a = new Animal();
        Mammal m = new Mammal();
        Dog d = new Dog();
        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
  }
}
```

```
//Output:
true
true
true
```

Inheritance (HAS-A)

Object Composition (HAS-A relationship)

Vehicle has a motor.
Dog has a tail.



Inheritance

Object Composition (HAS-A relationship)

```java
package Unit4.relationships;
class Car {
    // Methods implementation and class/Instance members
    private String color;
    private int maxSpeed;

    public void setColor ( String color ) {
        this.color = color;
    }

    public void setMaxSpeed ( int maxSpeed ) {
        this.maxSpeed = maxSpeed;
    }

    public void carInfo () {
        System.out.println("Car Color= " + color + " Max Speed= " + maxSpeed);
    }
}

class Maruti extends Car {
    //inherits all methods from Car (except final and static)
    //Maruti can also define all its specific functionality
    public void MarutiStartDemo () {
        Engine MarutiEngine = new Engine();
        MarutiEngine.start();
    }
}
```

```java
package Unit4.relationships;
public class RelationsDemoHAS_A {
    public static void main ( String[] args ) {
        Maruti myMaruti = new Maruti();
        myMaruti.setColor("RED");
        myMaruti.setMaxSpeed(180);
        myMaruti.carInfo();
        myMaruti.MarutiStartDemo();
    }
}
```

```java
package Unit4.relationships;
public class Engine {
    public void start () {
        System.out.println("Engine Started:");
    }

    public void stop () {
        System.out.println("Engine Stopped:");
    }
}
```

Subclasses and Superclasses

Extending Classes:

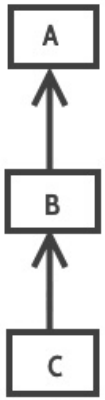
1. Group common attributes and methods
2. The keyword extends connects a subclass to a superclass via Inheritance
3. A superclass cannot use its subclass' features
4. A subclass can use the features of a superclass

Types of inheritance

Single Inheritance: refers to a child and parent class relationship where a class extends the another class.

Single Inheritance

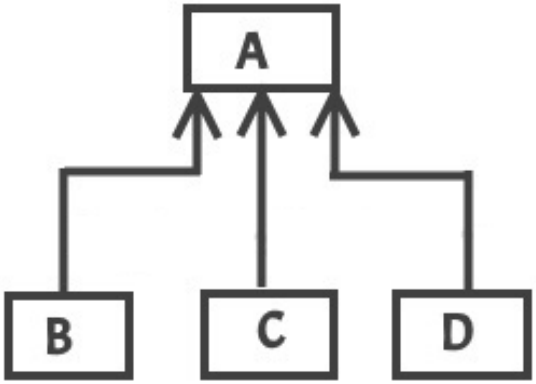
Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.



Multilevel Inheritance

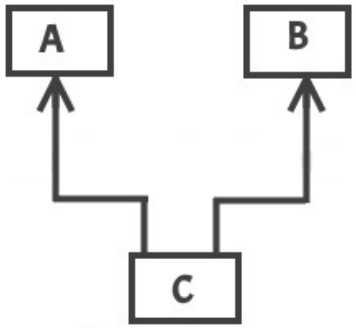
Types of inheritance

Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.



Hierarchical Inheritance

Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example class C extends both classes A and B. Java doesn't support multiple inheritance,



Multiple Inheritance

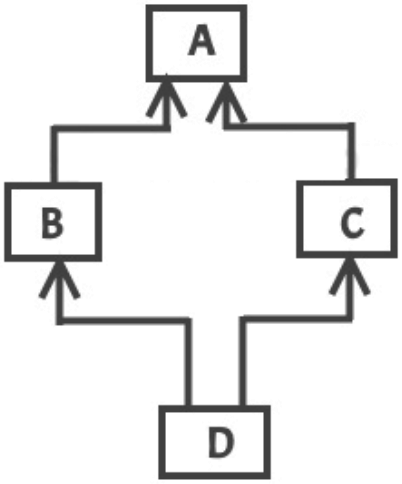
Assignment

Why Java doesn't support multiple inheritance?

- C++ , Commonly other languages supports multiple inheritance.
- While java doesn't support it. Java doesn't allow multiple inheritance to **avoid the ambiguity** caused by it.
- One of the example of such problem is the **diamond problem** that occurs in multiple inheritance.

What is diamond problem?

We will discuss this problem with the help of the diagram: which shows multiple inheritance as Class D extends both classes B & C. Now lets assume we have a method in `class A` and `class B` & `C` overrides that method in their own way. **Wait!! here the problem comes** – Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). **Ambiguity**. That's the main reason why Java doesn't support multiple inheritance.



Hybrid Inheritance

Assignment

Why Java doesn't support multiple inheritance?

Is multiple inheritance allowed in Java?

- Multiple inheritance faces problems when there exists a method with the same signature in both the superclasses.
- Due to such a problem, java does not support multiple inheritance directly, but the similar concept can be achieved using interfaces.
- A class can implement multiple interfaces and extend a class at the same time.

Some Important points :

- Interfaces in java are a bit like the class but with a significantly different.
- An Interface can only have method signatures field and a default method.
- The class implementing an interface needs to declare the methods (not field)
- You can create a reference of an interface but not the object
- Interface methods are public by default

Assignment

Solved By Interface ; Java doesn't support multiple inheritance?

Can we implement more than one interfaces in a class

Yes, we can implement more than one interfaces in our program because that doesn't cause any ambiguity(see the explanation below).

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always **abstract** by default, which doesn't let them give their implementation (or method definition) in interface itself.

```
interface X {
public void myMethod();
}
interface Y {
public void myMethod();
}
class JavaExample implements X, Y {
public void myMethod() {
System.out.println("Implementing more than one interfaces");
}
public static void main(String args[]){
JavaExample obj = new JavaExample();
obj.myMethod();
}
}
```

Output:

```
Implementing more than one interfaces
```

Super keyword in java

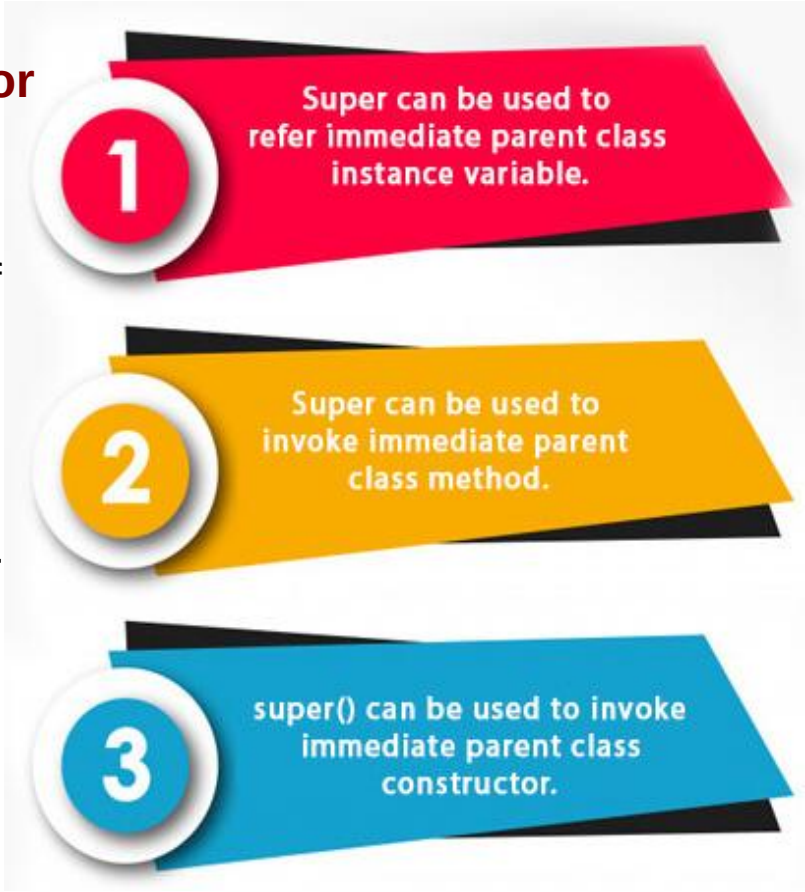
The super keyword refers to superclass (parent) objects.

The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.

The super keyword in Java is used in subclasses to access superclass members (attributes, methods and constructors).

Usage of Java super Keyword

1. Access **Attributes** of the Superclass
2. Access Overridden **Methods** of the superclass
3. Use of super() to access superclass **constructor**

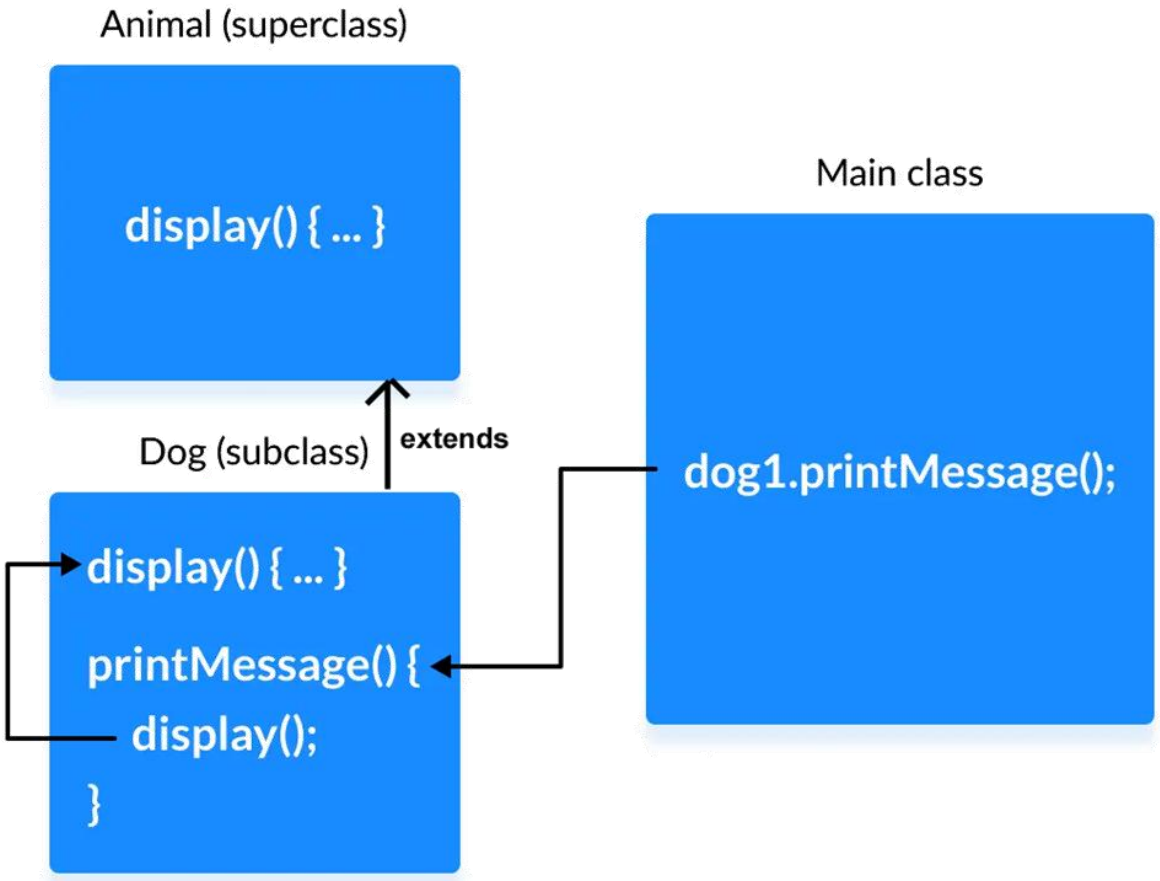


1. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
2. To call methods of the superclass that is overridden in the subclass.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

Usage of Java super Keyword

1. Access Overridden **Methods** of the superclass

Example 1: Method overriding



Usage of Java super Keyword

1. Access Overridden **Methods** of the superclass

Example 1: Method overriding

```
class Animal2 {
    // overridden method
    public void display () {
        System.out.println("I am an animal");
    }
}

class Dog1 extends Animal2 {
    // overriding method
    public void display () {
        System.out.println("I am a dog");
    }
    public void printMessage () {
        display();
    }
}

class Main {
    public static void main ( String[] args ) {
        Dog1 dog1 = new Dog1();
        dog1.printMessage();
    }
}
```

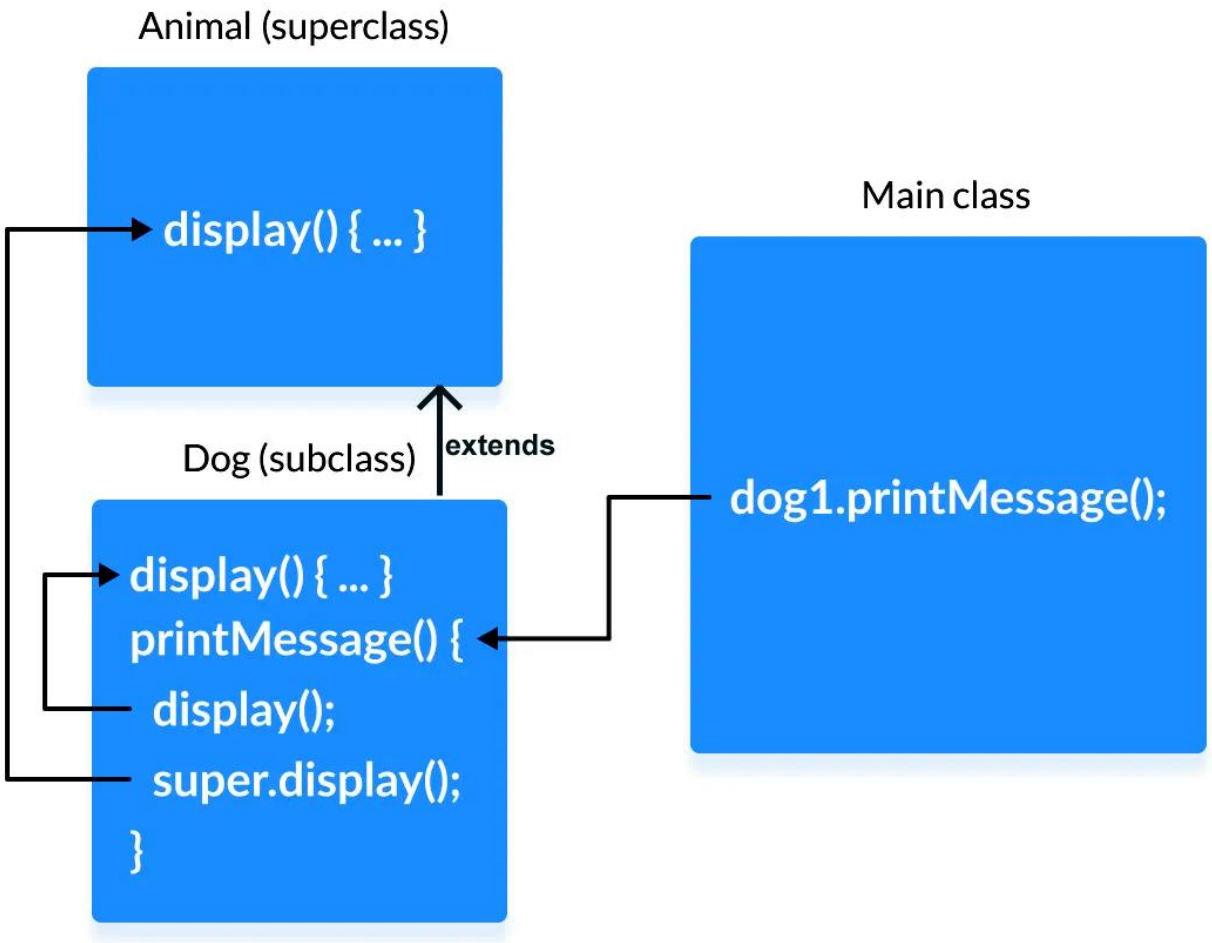
Output

```
I am a dog
```

Usage of Java super Keyword

1. Access Overridden **Methods** of the superclass

Example 2: Method overriding



Usage of Java super Keyword

1 Access Overridden **Methods** of the superclass

Example 2: Method overriding

```
class Animal2 {
    // overridden method
    public void display () {
        System.out.println("I am an animal");
    }
}

class Dog1 extends Animal2 {
    // overriding method
    public void display () {
        System.out.println("I am a dog");
    }
    public void printMessage () {
        // this calls overriding method
        display();
        // this calls overridden method
        super.display();
    }
}
class Main {
    public static void main ( String[] args ) {
        Dog1 dog1 = new Dog1();
        dog1.printMessage();
    }
}
```

Output

I am a dog I am an animal

Overriding Methods

- ✓ Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.
- ✓ Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- ✓ In this case the method in parent class is called ***overridden method*** and the method in child class is called ***overriding method.***

```
class Human{
    //Overridden method
    public void eat()
    {
        System.out.println("Human is eating");
    }
}
class Boy extends Human{
    //Overriding method
    public void eat(){
        System.out.println("Boy is eating");
    }
    public static void main( String args[]) {
        Human obj = new Boy();
        //This will call the child class version of eat()
        obj.eat();
    }
}
```

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.

Output:

```
Boy is eating
```

Dynamic method dispatch

- Dynamic method dispatch is also known as run time polymorphism.
- It is the process through which a call to an overridden method is resolved at runtime.
- This technique is used to resolve a call to an overridden method at runtime rather than compile time.
- To properly understand Dynamic method dispatch in Java is based on upcasting.

Upcasting :

- It is a technique in which a superclass reference variable refers to the object of the subclass.

```
class Animal{}
class Dog extends Animal{}
Animal obj =new Dog();//upcasting
```

Advantages of dynamic method dispatch

1.It allows Java to support overriding of methods, which are important for run-time polymorphism.

2.It allows a class to define methods that will be shared by all its derived classes, while also allowing these sub-classes to define their specific implementation of a few or all of those methods.

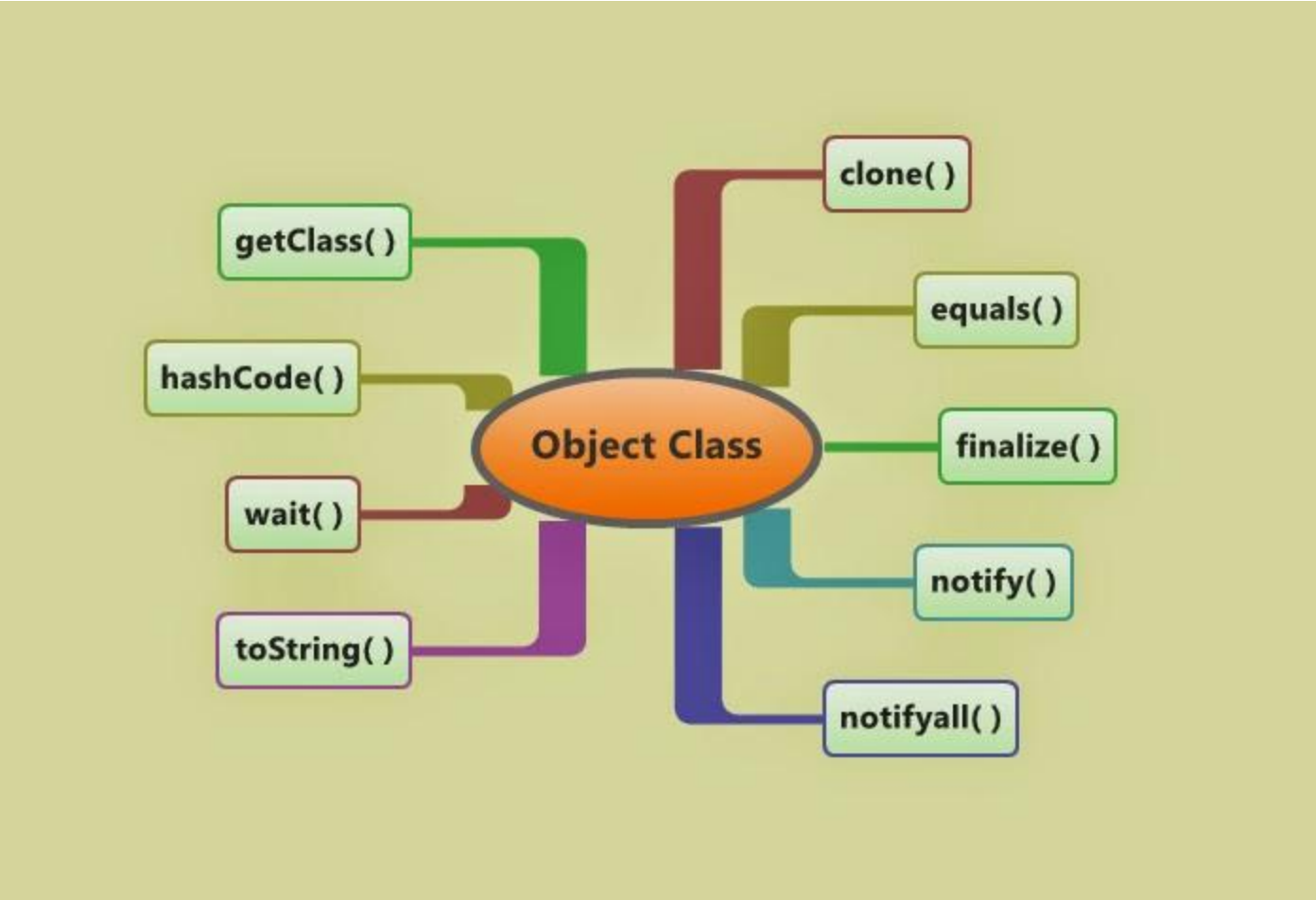
3.It allows subclasses to incorporate their own methods and define their implementation.

Dynamic method dispatch *is the mechanism in which a call to an overridden method is resolved at run time instead of compile time.*

Dynamic Method Dispatch

```java
class ABC{
    //Overridden method
    public void disp()
    {
        System.out.println("disp() method of parent class");
    }

}
class Demo extends ABC{
    //Overriding method
    public void disp(){
        System.out.println("disp() method of Child class");
    }
    public void newMethod(){
        System.out.println("new method of child class");
    }
    public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object
         * then in this case overridden method (the method of parent class)
         *  is called.
         */
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object
         * then the overriding method (method of child class) is called.
         * This is called dynamic method dispatch and runtime polymorphism
         */
        ABC obj2 = new Demo();
        obj2.disp();
        // obj2.newMethod(); error Unresolved compilation, newMethod() is undefined for the type ABC
    }
}
```

The Object class: Object Class Methods



The Object class: Object Class Methods

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether two objects have the same state
`protected void finalize()`	called during garbage collection
`public Class<?> getClass()`	info about the object's type
`public int hashCode()`	a code suitable for putting this object into a hash collection
`public String toString()`	text representation of the object

`public void notify()`
`public void notifyAll()`
`public void wait()`
`public void wait(...)`
	methods related to concurrency and locking (seen later)

https://moonlightenoch.wordpress.com/2012/02/24/java-se-review-generics-and-collections/

Generics and Collections see later

The Object class: Object Class Methods

1. protected Object clone() Method

```
import java.util.*;
public class ObjectClassClone {
    public static void main(String[] args) {
        Date date = new Date();
        System.out.println(date.toString());
        Date date2 = (Date) date.clone();
        System.out.println(date2.toString());
    }
}
```

Output:
Tue Oct 04 08:26:52 NPT 2022
Tue Oct 04 08:26:52 NPT 2022

The Object class: Object Class Methods

2. boolean equals(Object obj)

```
public class ObjectClassEquals {
    public static void main ( String[] args ) {
        // get an integer, which is an object
        Integer x = new Integer(50);

        // get a float, which is an object as well
        Float y = new Float(50f);

        // check if these are equal,which is
        // false since they are different class
        System.out.println("" + x.equals(y));

        // check if x is equal with another int 50
        System.out.println("" + x.equals(50));
    }

}
```

Output:
false
true

The Object class: Object Class Methods

2. boolean equals(Object obj)

```
public class ObjectClassEquals {
    public static void main ( String[] args ) {
        // get an integer, which is an object
        Integer x = new Integer(50);

        // get a float, which is an object as well
        Float y = new Float(50f);

        // check if these are equal,which is
        // false since they are different class
        System.out.println("" + x.equals(y));

        // check if x is equal with another int 50
        System.out.println("" + x.equals(50));
    }

}
```

Output:
false
true

The Object class: Object Class Methods

4. Class<?> getClass() Method

```
package Unit4;
class Person {
    private String firstName;

    public static void main ( String[] args ) {
        Person person = new Person();
        System.out.println(person.getClass());
    }

}
```

Output:
class Unit4.Person

The Object class: Assignment

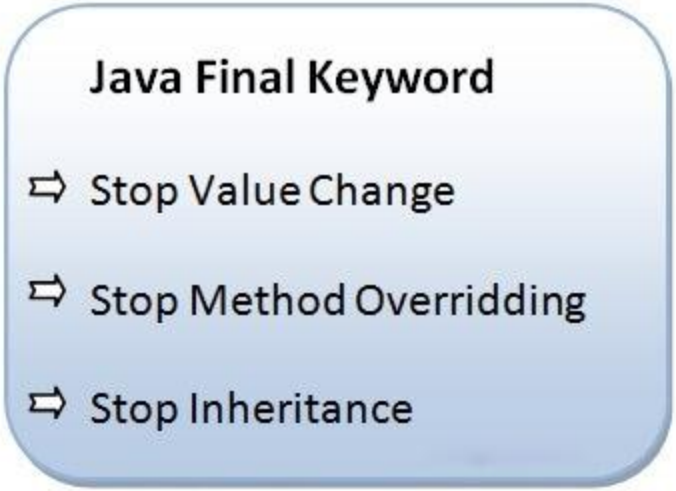
Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Final Classes

- 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



Final Classes

1) final variable

final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
class Bike{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike obj= new  Bike();
 obj.run();
 }
}//end of class
```

Output:Compile Time Error

Final Classes

2) final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output:Compile Time Error

Final Classes

3) final class

We cannot extend a final class. Consider the below example:

```
final class Bike {

}

class Honda1 extends Bike{
  void run() {
            System.out.println("running safely with 100kmph");
            }

  public static void main(String args[]){
            Honda1 honda= new Honda1();
            honda.run();
  }
}
```

Output:Compile Time Error

Final Classes Assignment

Q) Is final method inherited?
Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike {
    final void run () {
        System.out.println("running...");
    }
}

class Honda2 extends Bike {
    public static void main ( String args[] ) {
        Bike b = new Honda2();
        b.run();
    }

}
```

Output:

```
running...
```

Final Classes Assignment

Points to Remember:

1) A **constructor** cannot be declared as final.
2) Local final variable must be initializing during declaration.
3) All variables declared in an **interface** are by default final.
4) We cannot change the value of a final variable.
5) A final method cannot be overridden.
6) A final class not be inherited.
7) If method parameters are declared final then the value of these parameters cannot be changed.
8) It is a good practice to name final variable in all CAPS.
9) final, **finally** and finalize are three different terms. finally is used in exception handling and finalize is a method that is called by JVM during **garbage collection**.

Package

- A package is used to group related classes.
- Packages help in avoiding name conflicts

Example of Build-in:

```
import java.util.Scanner
```

Here:
→ **java** is a top level package
→ **util** is a sub package
→ and **Scanner** is a class.

There are two types of packages :

1. Build-in packages - java API
2. User-defined packages - Custom packages

Using a java package

Import keyword is used to import packages in the java program.
Example :

```
import java.lang.*;  - import
import java.lang.String; - import string from java.lang
String s = new java.lang.String("Hello"); - use without  importing
```

Advantages

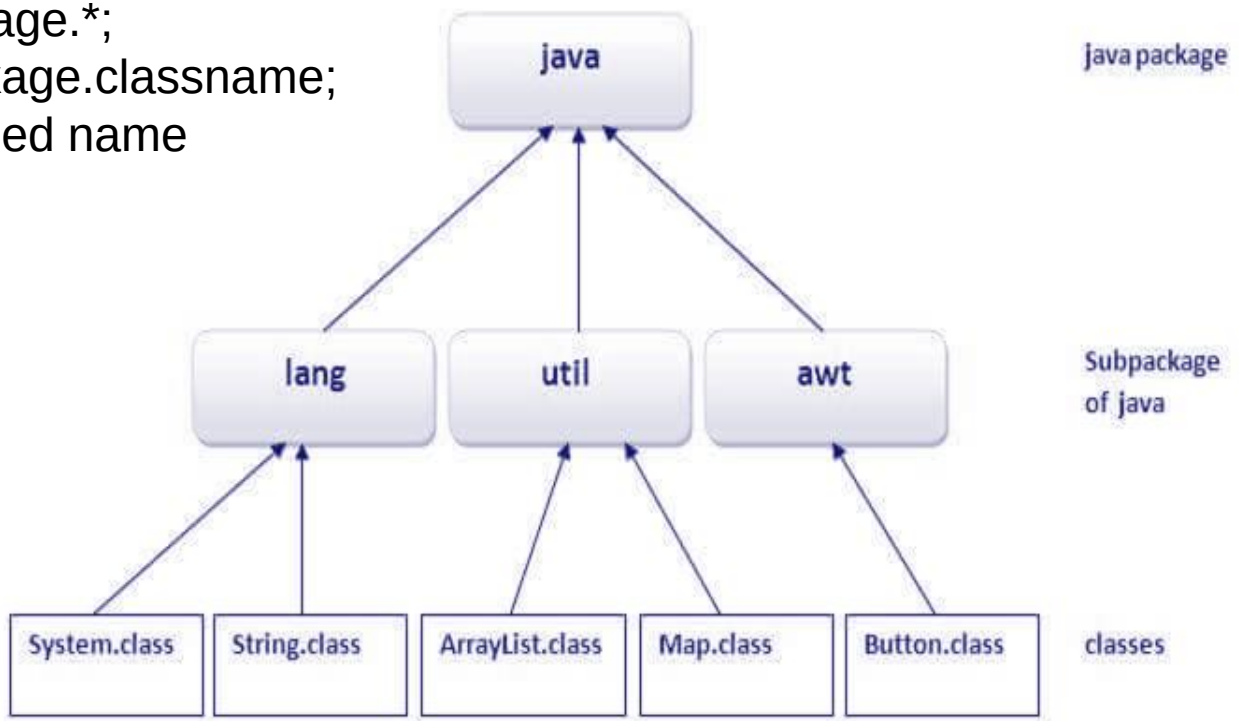
1. Reusability
2. Better Organization
3. Name Conflicts
4. Classes and interfaces separate, easily maintained
5. Access Protection

Package: Build-in

Package is used to make classes and interfaces as single unit.

<u>3 ways to access package from outside package:</u>

i. import package.*;
ii. Import package.classname;
iii. Fully qualified name



There different built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

To avoid naming conflicts packages are given names of the domain name of the company in reverse Ex: np.com.spm, com.microsoft, com.infosys etc.

Package: Build-in

```
package spm.com.np;
import java.util.Scanner;
//import java.util.*;
public class packagesbuildin {
       public static void main(String[] args) {
              Scanner sc = new Scanner(System.in);
      // java.util.Scanner sc = new java.util.Scanner(System.in);
              int a = sc.nextInt();
              System.out.println("Taking input as " + a);
   }
}
```

Package: Create into Pakage

Create folder "pakagesdemo" & Make "Sum.java"

```
package pakagesdemo;
public class Sum {
    public int SumFun(int a, int b){
        int sum;
        sum = a + b;
        return sum;
    }
}
```

Inside folder "pakagesdemo" Make "Multiple.java"

```
package pakagesdemo;
public class Multiple {
    public int MultipleFun(int a, int b){
        int p;
        p = a * b;
        return p;
    }
}
```

Package: command

```
PS C:\Users\sital\Documents\Java\pakagesdemo> javac Sum.java
PS C:\Users\sital\Documents\Java\pakagesdemo> javac Multiple.java
PS C:\Users\sital\Documents\Java\pakagesdemo> javac -d . Sum.java
PS C:\Users\sital\Documents\Java\pakagesdemo> javac -d . *.java
```

All files

Current Folder

How to create a package in Java :

`javac Sum.java :` create Sum class

`javac –d . Sum.java :` creates a packages to current folder

// We can keep adding classes to current package

Note:

- We can also create inner packages by adding "*package.inner*" as the package name. *// folder.subfolder*
- These packages once created can be used by other classes.

Package: UsingPakage

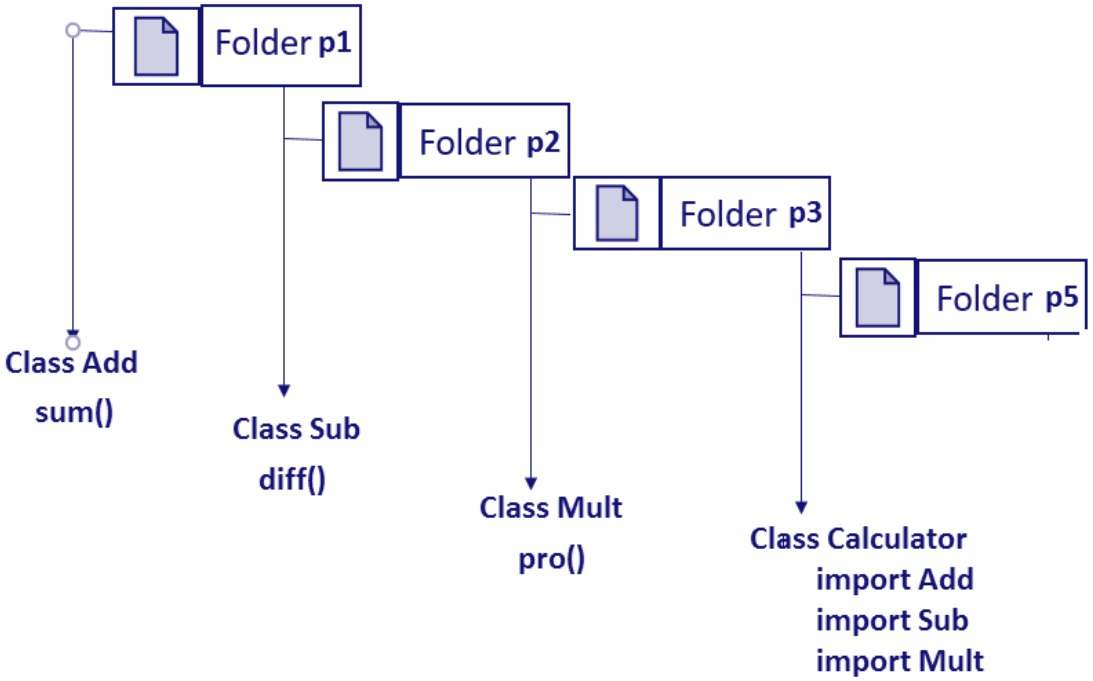
Outside folder "pakagesdemo" Make "UsingPakage.java"

```
import pakagesdemo.Multiple;
import pakagesdemo.Sum;

public class UsingPakage{
    public static void main(String[] args) {
        Multiple m = new Multiple();
        int mul = m.MultipleFun(5,6);
        System.out.println(mul);

        Sum s = new Sum();
        int sum = s.SumFun(5, 5);
        System.out.println(sum);
    }

}
```

Package: Assignment

Practice Set on Java Package

- Create three classes Add, Sub, Mul and Calculator into a package
- Use a build-in package in java to write a class which displays a message (by using sout) after taking input from the user
- Create a package in class with four package levels folder p1 , folder p2 , folder p3 & p5 with their respected method sum(), diff(), pro() and call all method into class Calculator.



https://ctaljava.blogspot.com/2022/10/practice-set-on-java-package.html

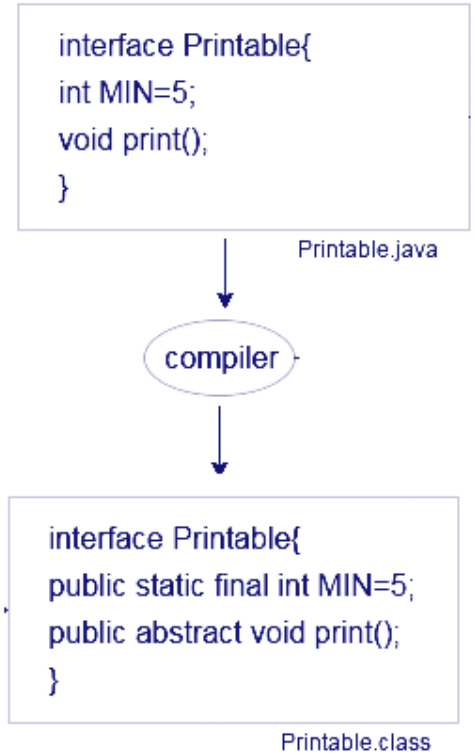
Interface

Defining an Interface

- Interface is a collection of method declarations.
- We can implement multiple inheritance using interface.
- Because interface consists only signatures followed by semi colon and parameter list they are implicitly abstract.
- Variables can be declared and initialized inside interface they are implicitly final and static.
- An interface method can't be final or static.
- An interface can be extended from another interface.

Declaration of interface:

Access interface name
{
Return type member-name1(parametelist);
Return type member-name2(parametelist);
.
.
.
Type finalvariablename=initialization;
}



Interface Assignment

Implementing and applying interfaces

- Just like a class in java is a collection of the related methods, an interface in java is a collection of abstract methods.
- The interface is one more way to achieve abstraction in Java.
- An interface may also contain constants, default methods, and static methods.
- All the methods inside an interface must have empty bodies except default methods and static methods.
- We use the **interface** keyword to declare an interface.
- There is no need to write **abstract** keyword before declaring methods in an interface because an interface is implicitly abstract.
- An interface cannot contain a constructor (as it cannot be used to create objects)
- In order to implement an interface, java requires a class to use the **implement** keyword.

Interface Assignment

Implementing and applying interfaces

Example 1: Java Interface

```
// create an interface
interface Language {
    void getName ( String name );
}

// class implements interface
class ProgrammingLanguage implements Language {

    // implementation of abstract method
    public void getName ( String name ) {
        System.out.println("Programming Language: " + name);
    }
}

class InterfaceMain1 {
    public static void main ( String[] args ) {
        ProgrammingLanguage language = new ProgrammingLanguage();
        language.getName("Java");
    }
}
```

Interface Assignment

Implementing and applying interfaces

Example 2: Find area of rectangle using Java Interface "Polygon"

```
interface Polygon {
    void getArea ( int length, int breadth );
}

// implement the Polygon interface
class Rectangle implements Polygon {

    // implementation of abstract method
    public void getArea ( int length, int breadth ) {
        System.out.println("The area of the rectangle is " + (length * breadth));
    }
}

class InterfaceMain {
    public static void main ( String[] args ) {
        Rectangle r1 = new Rectangle();
        r1.getArea(5, 6);
    }
}
```

Interface Assignment

Implementing and applying interfaces

Implementing Multiple Interfaces

```
interface A {
    // members of A
}

interface B {
    // members of B
}

class C implements A, B {
    // abstract members of A
    // abstract members of B

}
```

Extending an Interface

```
interface Line {
    // members of Line interface
}

// extending interface
interface Polygon extends Line {
    // members of Polygon interface
    // members of Line interface
}
```

Extending Multiple Interfaces

```
interface A {
    ...
}
interface B {
    ...
}

interface C extends A, B {
    ...
}
```

Interface Assignment

Implementing and applying interfaces

Example 3: abstract class and interface in Inheritance Java

```java
//Creating interface that has 4 methods
interface A {
    void a ();//bydefault, public and abstract

    void b ();

    void c ();
}
```

```java
//Creating abstract class that provides the
//implementation of one method of A interface
abstract class B implements A {
    public void c () {
        System.out.println("I am C");
    }
}
```

```java
//Creating subclass of abstract class, now we need to provide
// the implementation of rest of the methods
class M extends B {
    public void a () {
        System.out.println("I am a");
    }

    public void b () {
        System.out.println("I am b");
    }
}
```

```java
//Creating a test class that calls the methods
// of A interface
class InterfaceMain3 {
    public static void main ( String args[] ) {
        A a = new M();
        a.a();
        a.b();
        a.c();
    }
}
```

Interface Assignment

Implementing and applying interfaces

Example 4: interface with Inheritance Java

```
interface Inf1{
    public void method1();
}
interface Inf2 extends Inf1 {
    public void method2();
}
```

```
public class InterfaceMain4 implements Inf2{
    /* Even though this class is only implementing the
     * interface Inf2, it has to implement all the methods
     * of Inf1 as well because the interface Inf2 extends Inf1
     */
    public void method1(){
        System.out.println("method1");
    }
    public void method2(){
        System.out.println("method2");
    }
    public static void main(String args[]){
        Inf2 obj = new InterfaceMain4();
        //obj.method1();
        obj.method2();
    }
}
```