OOP Fundamentals

continued…

PACKAGE

What Is a Package?

A package is a namespace that organizes a set of RELATED classes and interfaces. It is similar to different folders on your computer.

Example - *Image based classes in one package, maths based in another package, general utility based classes in another, and so on...*

The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short.

```
        import          java.lang.*;
        import          java.util.*;
```

Example -

- a `String` object contains state and behavior for character strings;
- a `File` object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem;
- a `Socket` object allows for the creation and use of network sockets;

ACCESS SPECIFIERS in JAVA

What is Access Specifier?
A mechanism to set access levels for classes, variables, methods and constructors. The four access levels are:

//In increasing order of accessibility

• PRIVATE -	Visible to the same class only.
• DEFAULT -	Visible to the same package. No modifiers are needed.
• PROTECTED -	Visible to the package and all subclasses.
• PUBLIC -	Visible to the world.

```
public class Student {
        private String course;

        }
```

ACCESS SPECIFIERS in JAVA

```
class Test{
       public static void main(String[] args){
              Student ramesh = new Student();

              //To set the value of the course attribute.
              //INCORRECT
              ramesh.course="B.Tech";


              //To get the value of the course attribute.
              //This statement is INCORRECT
              System.out.println("Course name of ramesh is: "
              +ramesh.course);
       }
}
```

ACCESS SPECIFIERS in JAVA

What is Access Specifier?
A mechanism to set access levels for **classes, attributes, variables, methods and constructors**. The four access levels are:

//In increasing order of accessibility

- PRIVATE - Visible to the same class only.
- DEFAULT - Visible to the same package. No modifiers are needed.
- PROTECTED - Visible to the package and all subclasses.
- PUBLIC - Visible to the world.

```
public class Student {
        private String course;

        public void setCourse(String courseNew) {
                this.course = courseNew;
        }

        public String getCourse() {
                return this.course;
        }
}
```

ACCESS SPECIFIERS in JAVA

```
class Test{
       public static void main(String[] args){
              Student ramesh = new Student();

              //To set the value of the course attribute.
              //INCORRECT
              ramesh.course="B.Tech";

              //CORRECT
              ramesh.setCourse("B.Tech");



              //To get the value of the course attribute.
              //This statement is INCORRECT
              System.out.println("Course name of ramesh is: "
              +ramesh.course);

              //CORRECT statement
              System.out.println("Course name of ramesh is: "
              +ramesh.getCourse());
       }
}
```

this KEYWORD in JAVA

Within a method/constructor, this is a reference to the current object — the object whose method/constructor is being called.

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b){
        x = a;
        y = b;
    }
}
```

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

this KEYWORD in JAVA – Example 2

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation.

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }

    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

ENCAPSULATION

Technical Definition:

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods.

If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

ENCAPSULATION - EXAMPLE

```
1. class Student{
2.     private String name;
3.     public String getName(){
4.           return name;
5.     }
6.     public void setName(String newName){
7.           name = newName;
8.     }
9. }
10.class Execute{
11.    publi                            []){
12.
13.
14.          localName = s1.getName();
15.    }
16.}
```

The public methods are the access points to a class's private fields(attributes) from the outside class.

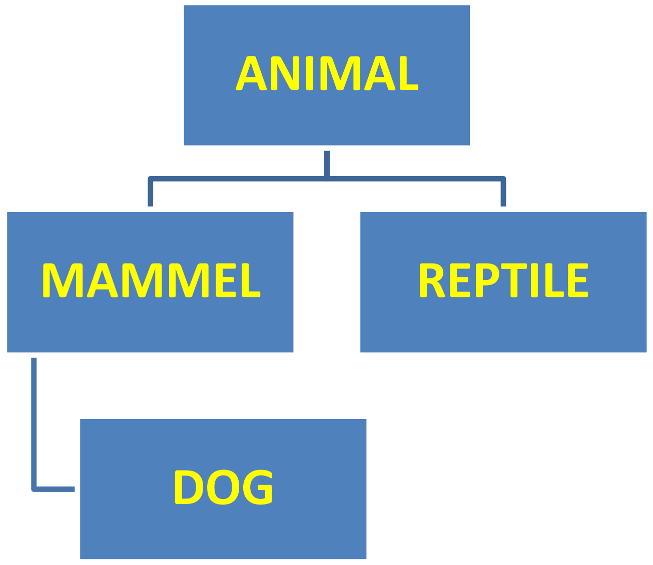
At line no14, we can not write `localName = s1.name;`

INHERITANCE

Technical Definition:

Inheritance can be defined as the process where one object (or class) acquires the properties of another (object or class).

With the use of inheritance the information is made manageable in a hierarchical order.

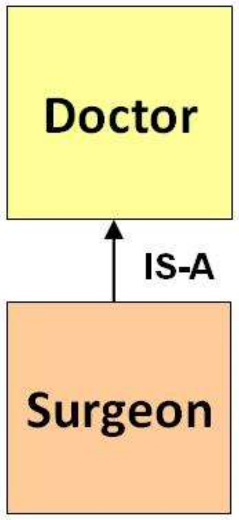


INHERITANCE

- In programming, inheritance is brought by using the keyword **EXTENDS**
- In theory, it is identified using the keyword **IS-A.**
- By using these keywords we can make one object acquire the properties of another object.

This is 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

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.

Alternatively, 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 – EXAMPLE – IS-A RELATIONSHIP

```
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);
   }
}
```

This would produce the following result:

```
true
true
true
```

INHERITANCE – EXAMPLE – HAS-A RELATIONSHIP

Determines whether a certain class HAS-A certain thing.

Lets us look into an example:

```
class Vehicle{}
class Speed{}
class Van extends Vehicle{
    private Speed sp;
}
```

This shows that class Van HAS-A Speed.

By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class Dog extends Animal, Mammal{};
```

This is wrong

OVERRIDING

The process of defining a method in child class with the same name & signature as that of a method already present in parent class.

- If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.
- Benefit: A subclass can implement a parent class method based on its requirement.
- In object-oriented terms, overriding means to override the functionality of an existing method.

OVERRIDING – EXAMPLE 1

```
class Animal{
    public void move(){
            System.out.println("Animals can move");
    }
}
class Dog extends Animal{
    public void move(){
            System.out.println("Dogs can walk and run");
    }
}
class TestDog{
    public static void main(String args[]){
            Animal a = new Animal();
            // Animal reference and object

            Animal b = new Dog();
            // Animal reference but Dog object
            a.move();// runs the method in Animal class
            b.move();//Runs the method in Dog class
    }
}
```

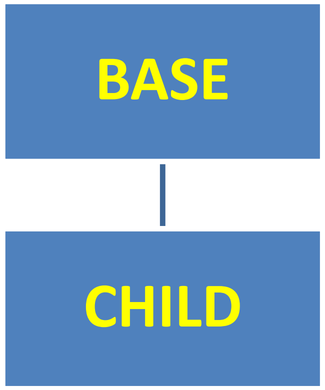
```
Animals can move
Dogs can walk and run
```

OVERRIDING – BASE REFERENCE AND CHILD OBJECT

Suppose there is a scenario that CHILD inherits BASE class, as shown in the figure.

Both BASE and CHILD classes would have their own methods, as well as some overridden methods, if any.

Category I: Methods present in BASE class only.
Category II: Methods present in CHLD class only.
Category III: Methods available in BASE but overridden in CHILD.



Now, if we create an object with `BASE` class reference and `CHILD` class object as:

```
    BASE ref = new CHILD();
```

Then, **`ref`** could access the methods belonging to Category I and Category III only.

OVERRIDING – EXAMPLE 1 - EXPLANANTION

In the above example, even though ***b*** is a type of Animal; it runs the ***move*** method in the Dog class.

REASON:

In compile time, the check is made on the reference type.

However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method **`move`**. Then, at the runtime, it runs the method specific for that object.

OVERRIDING – EXAMPLE 2

```
1.  class Animal{
2.  }
3.  class Dog extends Animal{
4.     public void bark(){
5.        System.out.println("Dogs can bark");
6.     }
7.  }
8.  public class TestDog{
9.     public static void main(String args[]){
10.       Animal a = new Animal(); // Animal reference and object
11.       Animal b = new Dog(); // Animal reference but Dog object
12.       b.bark();
13.    }
14. }
```

Result (ERROR):

```
TestDog.java:12: cannot find symbol
symbol  : method bark()
location: class Animal  b.bark();
```

RULES FOR METHOD OVERRIDING – PART 1

1. The argument list should be exactly the same as that of the overridden method.

2. The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.

3. The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.

4. Instance methods can be overridden only if they are inherited by the subclass.

5. A method declared final cannot be overridden.

6. A method declared static cannot be overridden but can be re-declared.

RULES FOR METHOD OVERRIDING – PART 2

7. If a method cannot be inherited, then it cannot be overridden.

8. A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.

9. A subclass in a different package can only override the non-final methods declared public or protected.

10. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.

11. Constructors cannot be overridden.

OVERRIDING: USING THE SUPER KEYWORD

```
class Animal{
      public void move(){
            System.out.println("Animals can move");
      }
}
class Dog extends Animal{
      public void move(){
            super.move(); // invokes the super class method
            System.out.println("Dogs can walk and run");
      }
}
class TestDog{
      public static void main(String args[]){
            Animal b = new Dog();
            // Animal reference but Dog object
            b.move();
            //Runs the method
      }
}
```

When invoking a superclass version of an overridden method the super keyword is used.

This would produce the following result:

```
Animals can move
Dogs can walk and run
```

POLYMORPHISM

POLYMORPHISM = 1 METHOD/OBJECT HAVING MANY FORMS/ROLES

CASE I: METHOD POLYMORPHISM

We can have multiple methods with the same name in the same / inherited / extended class.

There are three kinds of such polymorphism (methods):

1. **Overloading:** Two or more methods with different signatures, in the same class.
2. **Overriding:** Replacing a method of BASE class with another (having the same signature) in CHILD class.
3. Polymorphism by implementing **`Interfaces`**.

POLYMORPHISM – METHOD OVERLOADING

```
class Test {
    public static void main(String args[]) {
        myPrint(5);
        myPrint(5.0);
    }

    static void myPrint(int i) {
        System.out.println("int i = " + i);
    }

    static void myPrint(double d) {
        System.out.println("double d = " + d);
    }
}
```

Please note that the signature of overloaded method is different.

OUTPUT:

```
int i = 5
double d = 5.0
```

POLYMORPHISM – CONSTRUCTOR OVERLOADING

This example displays the way of overloading a constructor and a method depending on type and number of parameters.

```
class MyClass {
    int height;
    MyClass() {
        System.out.println("bricks");
        height = 0;
    }
    MyClass(int i) {
        System.out.println("Building new House that is "
        + i + " feet tall");
        height = i;
    }
    void info() {
        System.out.println("House is " + height
        + " feet tall");
    }
    void info(String s) {
        System.out.println(s + ": House is ‶ + height + " feet 
tall");
    }
}
```

POLYMORPHISM – CONSTRUCTOR OVERLOADING

```
public class MainClass {
    public static void main(String[] args) {
            //Calling of Overloaded constructor:
            MyClass t = new MyClass(0);

            t.info();
            //Calling of Overloaded method
            t.info("overloaded method");

           //Calling of DEFAULT constructor
            MyClass x = new MyClass();
    }
}
```

Result:
Building new House that is 0 feet tall.
House is 0 feet tall.
Overloaded method: House is 0 feet tall.
bricks

POLYMORPHISM

CASE II: OBJECT POLYMORPHISM

Example: The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Note: In the below statement

```
BASE ref = new CHILD();
```

NOTE: **ref** is the reference.

- Any Java object that can pass more than one IS-A test is considered to be polymorphic.
- The reference variable can be reassigned to other objects provided that it is not declared **final**.
- The type of the reference variable would determine the methods that it can invoke on the object.

OBJECT POLYMORPHISM - EXAMPLE

```
public class Animal{
      int a1;
      void am1(){...}
}
public class Deer extends Animal {
      int d1;
      void dm1(){....}
}

public class Execute{
      public static void main(String args[]){
            Deer d = new Deer();
            Animal a = d;

      }
}
```

Here, there are two references a & d.
And both references are pointing to same object.

Hence, 1 object can play many roles.