Object Oriented Programming

Unit 3

Creating Classes and Objects

Classes and Objects : implementations classes , class declaration , class modifiers Methods : methods , returning from methods, invoking methods , constructors, default constructor, implement constructors Overloading : Method overloading, Implement overloading static methods, Constructor Overloading Static members and Initialization Blocks: Static Class members, Implement static variable, Static Methods, Instance Initialization Block, Implement garbage collection, Packages, Class visibility, Access Levels, Implement package and import

Classes

- **A class is a set of data-members and member-functions.**
- A class defines composition character of an object.
- A class defines a **user-defined data type** which can be used to create objects of that type. Thus, a **class is a template for an object**.
- **Syntax:**

```
class class_name
 {
      data members
           and
     member functions
 }
```

- A **class** is declared by using **class** keyword and its members are declared and defined within the curly braces. Unlike C++, java class do not terminate with semi-colon(;).

Classes

Syntax of class:

```
class classname
{       type instance-variable;
        type methodname1(parameter-list)
        {
                // body of method
        }
        type methodname2(parameter-list)
        {
                // body of method
        }
}
```

Members of a class

- **Data members**
  - Data members are **variables** declared inside a class-for that class. Scope of data members is up-to that class only and all the member-functions of that class can directly access and modify values of data-members.
  - Java does not generate garbage values therefore it is necessary to initialize or assign values of data members before using them.
  - Variables of a class are also called as **instance variables**.

- **Member functions**
  - Member functions are **user defined functions**, defined in the class which contains actual code to process on.
  - These member functions have all logical statements to be executed. Member functions also have local variables whose scope is within that definition.
  - These member functions can have argument list and return-type as normal methods.

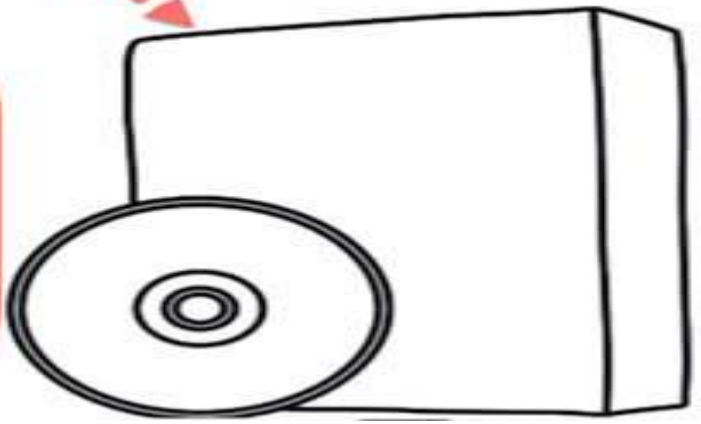
Members of a class Example

Example :

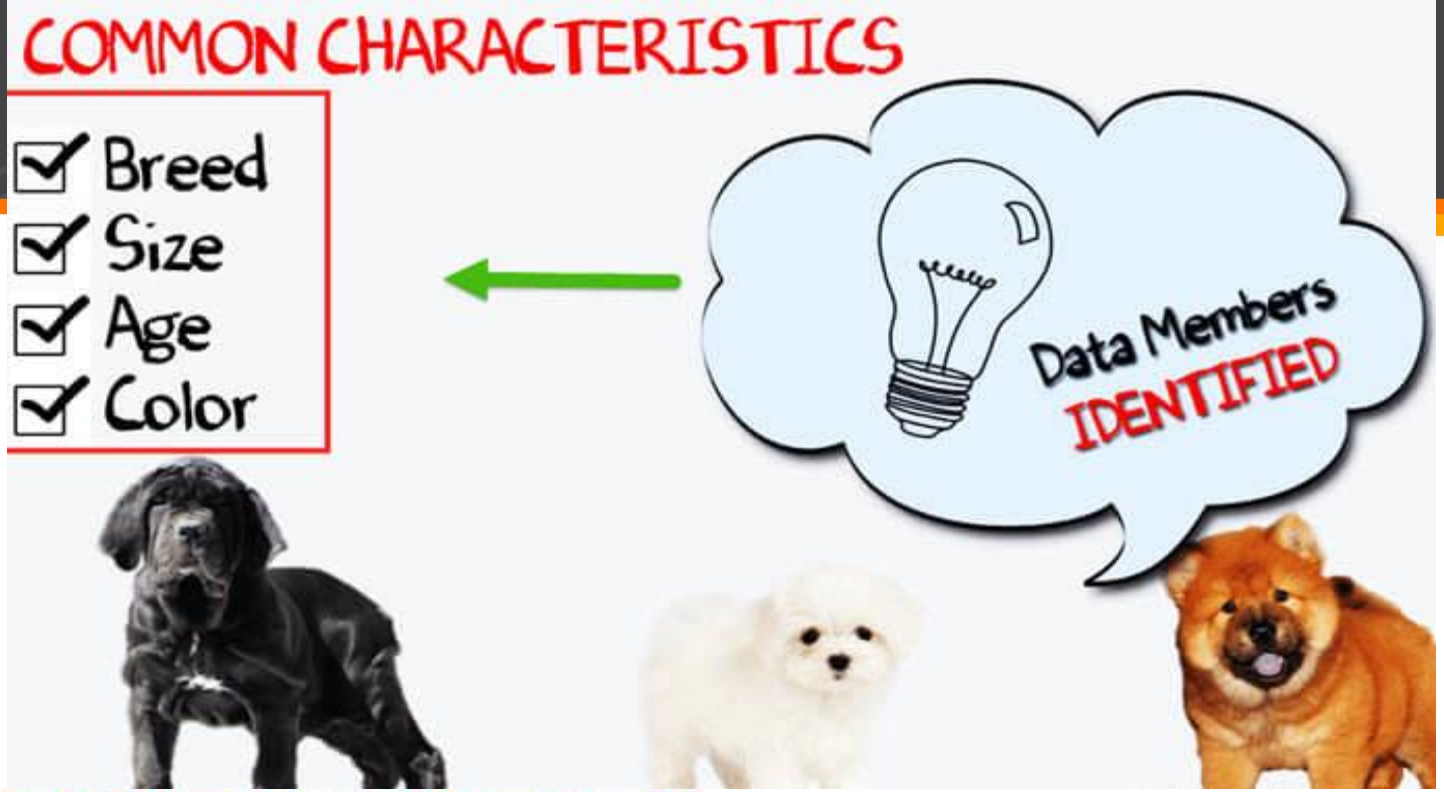
```
public class MyPoint
{
    int x = 0;
    int y = 0;
     void displayPoint()
    {
        System.out.println("Printing the coordinates");
        System.out.println(x + " " + y);
    }

    public static void main(String args[])
    {
        MyPoint obj;            // declaration
        obj = new MyPoint();            // allocation of memory to an object
        obj.x=10;               //access data member using object.
        obj.y=20;
        obj.displayPoint();     // calling a member method
    }
}
```

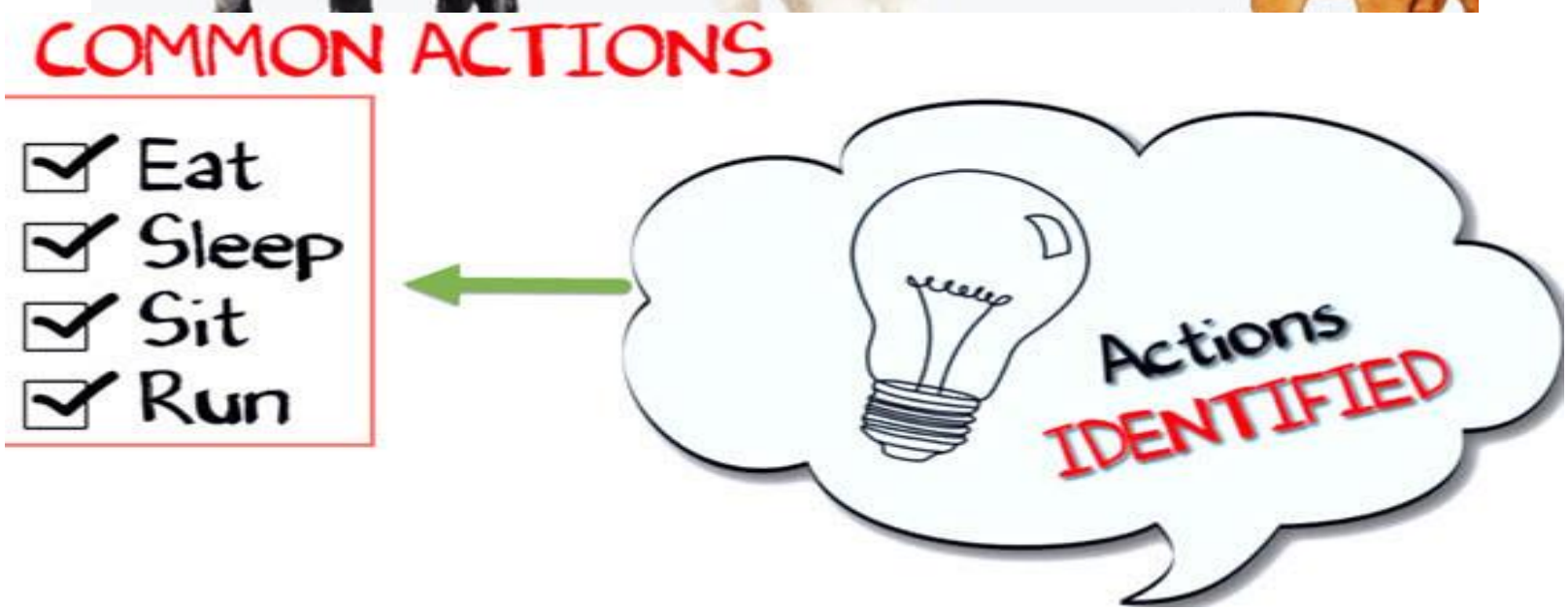
Pet management system

How to convert real life entities dogs into software objects?

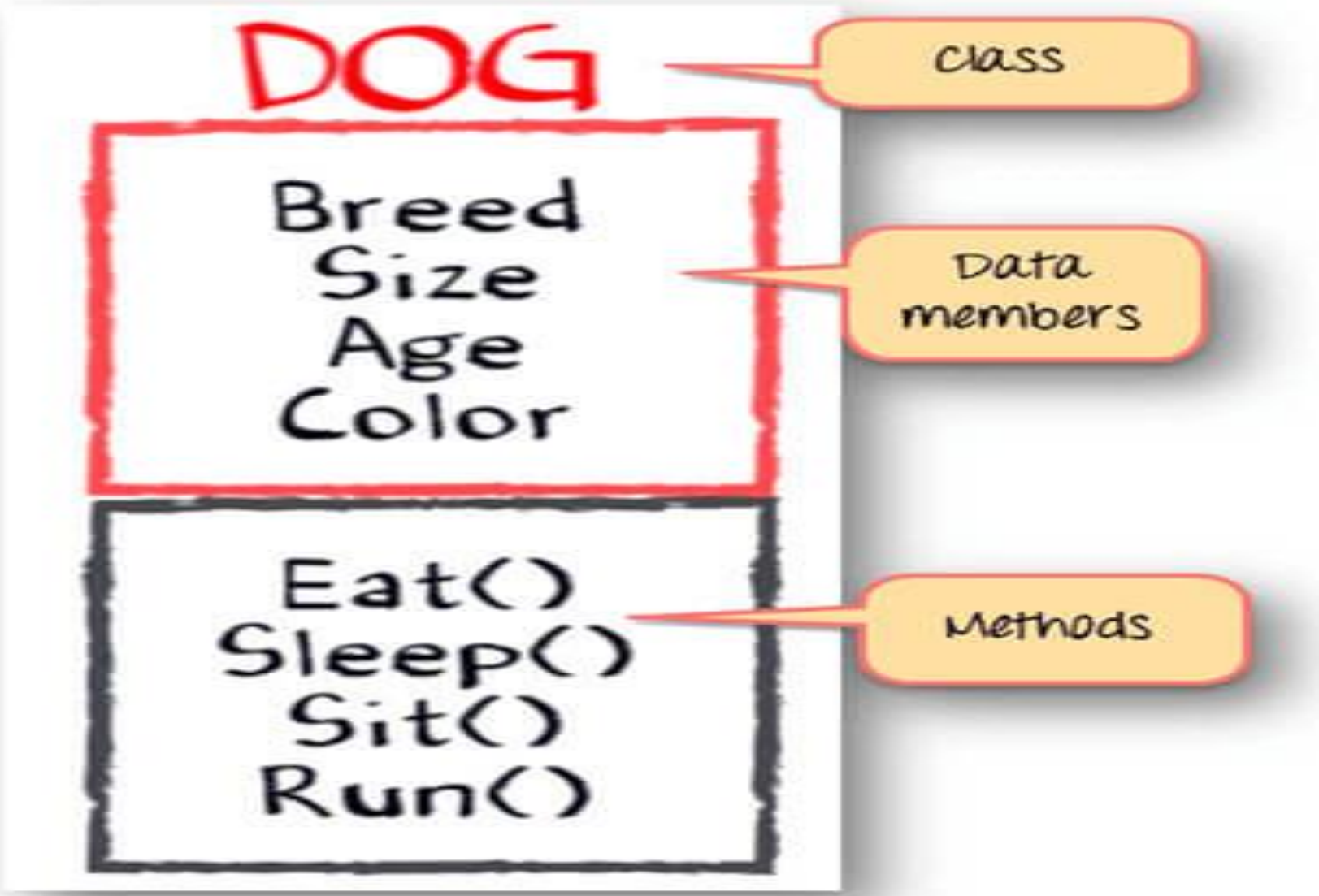


Spot the differences

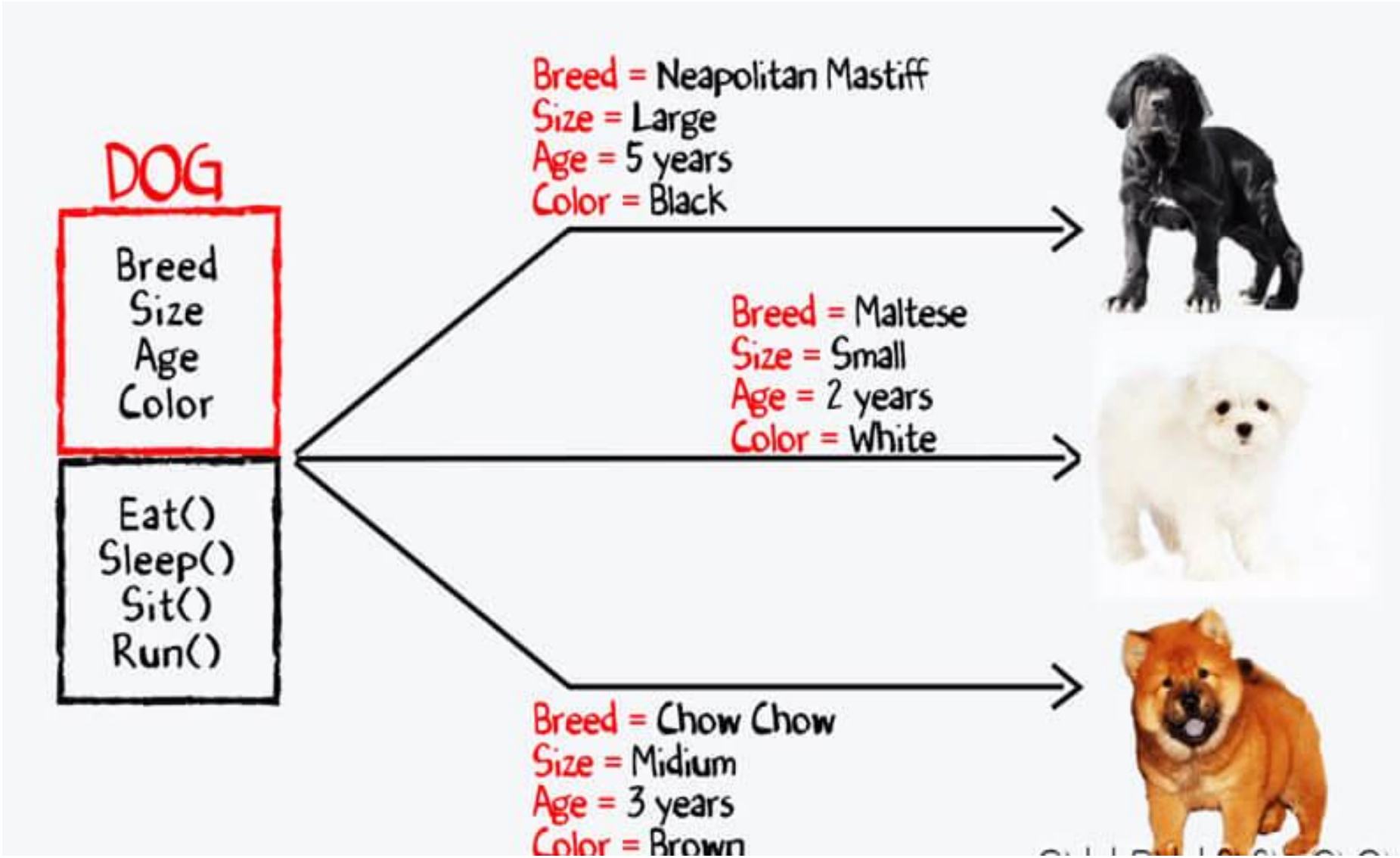

COMMON CHARACTERISTICS
Breed
Size
Age
Color
Data Members
IDENTIFIED


COMMON ACTIONS
Eat
Sleep
Sit
Run
Actions
IDENTIFIED

Pet management system



Pet management system



Access Specifier

- Java Access Specifiers (also known as Visibility Specifiers) regulate **access to classes, data members and methods in Java**.
- These specifiers determine whether a data member or method in a class, can be used or invoked by another method in another class or sub-class.

- Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programming.
- There are 4 types of java access modifiers:

- **private**
- **default**
- **protected**
- **public**

Access Specifier

1. private specifier

- Private Specifiers achieves the lowest level of accessibility.
- **private methods and fields can only be accessed within the same class to which the methods and fields belong.**
- private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier.
- Using Private Specifier we can **achieve encapsulation** and hide data from the outside world.

2. public specifier

- Public Specifiers achieves the highest level of accessibility.
- Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.

Access Specifier: private

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Access Specifier: public

```
class Hello
{
public int a=20;
public void show()
{
System.out.println("Hello java");
}
}

public class Demo
{
 public static void main(String args[])
 {
   Hello obj=new Hello();
   System.out.println(obj.a);
   obj.show();
 }
}
```

Access Specifier

3. protected specifier

- Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package of the protected members' class.
- The protected access specifier cannot be applied to class and interfaces.

4. Default (no specifier)

- When you don't set access specifier for the element, it will follow the default accessibility level. There is no default specifier keyword.
- Classes, variables, and methods can be default accessed. Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.
- Note: Default is not a keyword (like public, private, protected are keyword)

Access Specifier: protected

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Access Specifier: default

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

Declaring objects of a class

- **Object is copy of class** which occupies memory.
- **Syntax to declare an object:**

class_name object_name= new class_name(); //declaring and instantiating the object

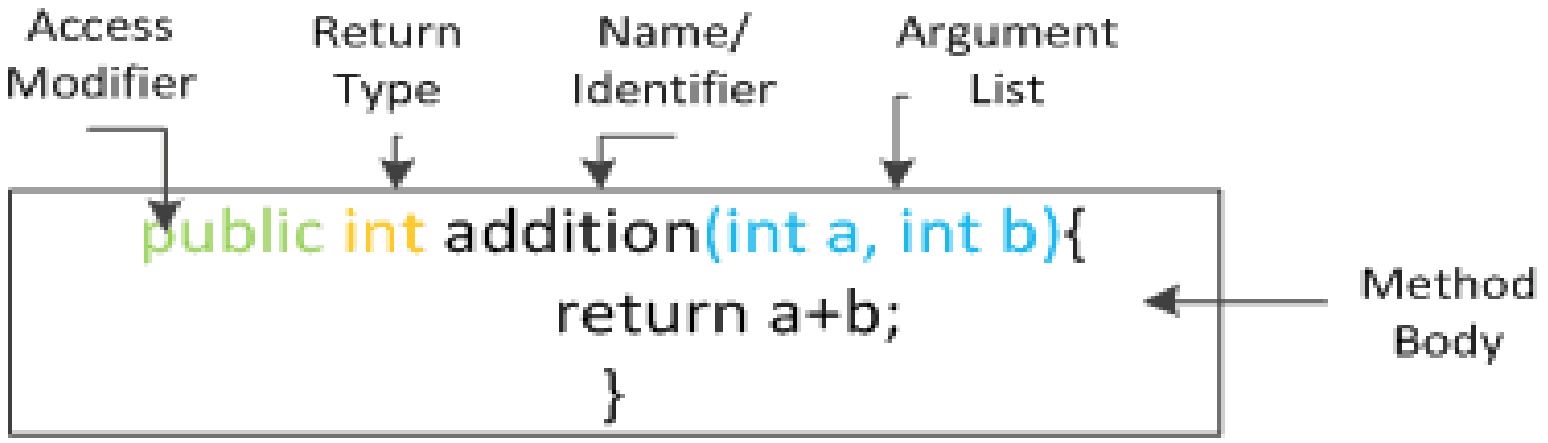
OR

class_name object_name; //declaring the object

object_name=new class_name(); // instantiating the object

Class Methods

- A method is a program module that contains a series of statements that carry out a task.
- To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method.
- In Java, every method must be part of some class which is different from languages like C, C++ and Python.
- Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times. The syntax to declare method is given below.



Accessing a Method

- **Syntax:**

```
modifier dataType methodName (methodParameters)
{
    //method header
   //statements that execute when called
}

//method call within the class:
          methodName(passedParameters);

//method call outside of the class, access using an object instance
          classInstance.methodName(passedParameters);
```

Method returning a value & Method Arguments

```
//Invoke (call) the method
int number1 = 25;
int number2 = 47;
int sum = add(number1, number2);

//Method definition
public int add(int x, int y)
{
    return (x + y);
}
```

actual parameters (or arguments)

formal parameters

Method returning a value & Method Arguments

```
import java.io.*;
public class method1test
{
    public static void main(String[ ] args)
    {
        int test1 = 12, test2 = 10;
       System.out.println("The minimum of " + test1 + " and " + test2
                                          + " is " + min(test1,test2) + ".");
    }
       public static int min(int x, int y)
            {
                int minimum;                    // local variable
                if (x < y)
                     minimum = x;
                else
                     minimum = y;
                return (minimum);     // return value to main
            }
}
```

Constructors

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the object is created, and memory is allocated for the object.
- It is a special type of method which is used to initialize the object.
- Every time **an object is created using new() keyword, at least one constructor is called. It calls a default constructor.**
- Rules for creating Java constructor:
  - Constructor name must be the **same as its class name**.
  - A Constructor must have **no explicit return type**.
  - A Java constructor **cannot be abstract, static, final, and synchronized**.

Constructors (Contd.)

- There are two types of constructors in Java:
  - **Default constructor (no-arg constructor)**
  - **Parameterized constructor**

❑ Default Constructor

- A constructor is called "Default Constructor" when it doesn't have any parameter.
- Syntax of default constructor:

```
classname()
{
}
```

Constructors (Contd.)

- **Default Constructor**

```
class Data
{
        Data()
        {
                System.out.println("Data class is created");
        }
        public static void main(String args[])
        {
                Data d=new Data();
        }
}
```

Constructors (Contd.)

❑ Parameterized Constructor

- A constructor which has a **specific number of parameters** is called a parameterized constructor.
- The **parameterized constructor is used to provide different values to the distinct objects**. However, you can provide the same values also.

Constructors (Contd.)

❑ **Parameterized Constructor**

```
class Student
{
   int id;
   String name;
   Student(int i,String n)
   {
          id = i;
          name = n;
   }
   void display()
   {
       System.out.println(id+" "+name);
   }
```

```
public static void main(String args[])
   {
    Student s1 = new Student(111,“ICT");
    Student s2 = new Student(222,“CS");
    s1.display();
    s2.display();
   }
}
```

Method Overloading

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

- **Different ways to overload the method**
  There are two ways to overload the method in java
- By changing number of arguments
- By changing the data type

Overloading static methods

1. Number of arguements

```
class Adder
{
  static int add(int a, int b)
 {
      return a+b;
 }
  static int add(int a, int b, int c)
 {
      return a+b+c;
  }
}
```

```
class TestOverloading2
{
  public static void main(String[] args)
 {
    System.out.println(Adder.add(11,11));
   System.out.println(Adder.add(11,11,11));
  }
}
```

Overloading static methods

2. Datatype of arguments

```
class Adder
{
  static int add(int a, int b)
 {
      return a+b;
 }
  static double add(double a, double b)
 {
      return a+b;
  }
}
```

```
class TestOverloading2
{
  public static void main(String[] args)
 {
    System.out.println(Adder.add(11,11));
   System.out.println(Adder.add(12.3,12.6));
  }
}
```

Constructor Overloading

```
class Student
{
   int id;
   String name;
   int age;
   Student(int i,String n)
   {
            id = i;
            name = n;
           age=19;
   }
    Student(int i, String n, int a)
   {
            id = i;
            name = n;
            age=a;
   }
```

```
 void display()
    {
         System.out.println(id+name+age);
    }
public static void main(String args[])
   {
    Student s1 = new Student(111,“ICT");
    Student s2 = new Student(222,“MI”,24);
    s1.display();
    s2.display();
   }
}
```

Static fields and methods

- 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 an instance of the class.
- The static can be:
  - Variable (also known as a class variable)
  - Method (also known as a class method)

Java static variable:

- If you declare any variable as static, it is known as a static variable.
- The static variable can also be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Static fields

```
class Student
  {
   int rollno;//instance variable
   String name;
   static String college ="IT";
   Student(int r,String n)
 {
   rollno = r;
   name = n;
   }
  void display ()
{
   System.out.println(rollno+" "+name+"
"+college);
 }
  }
```

```
public class TestStaticVariable1
{
 public static void main(String args[])
{
 Student s1 = new Student(111,"Aanya");
 Student s2 = new Student(222,"Arya");
  Student.college="PVC";
 s1.display();
 s2.display();
 }
}
```

Static methods

❑ Static methods

- If you apply static keyword with any method, it is known as **static method**.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Static method example

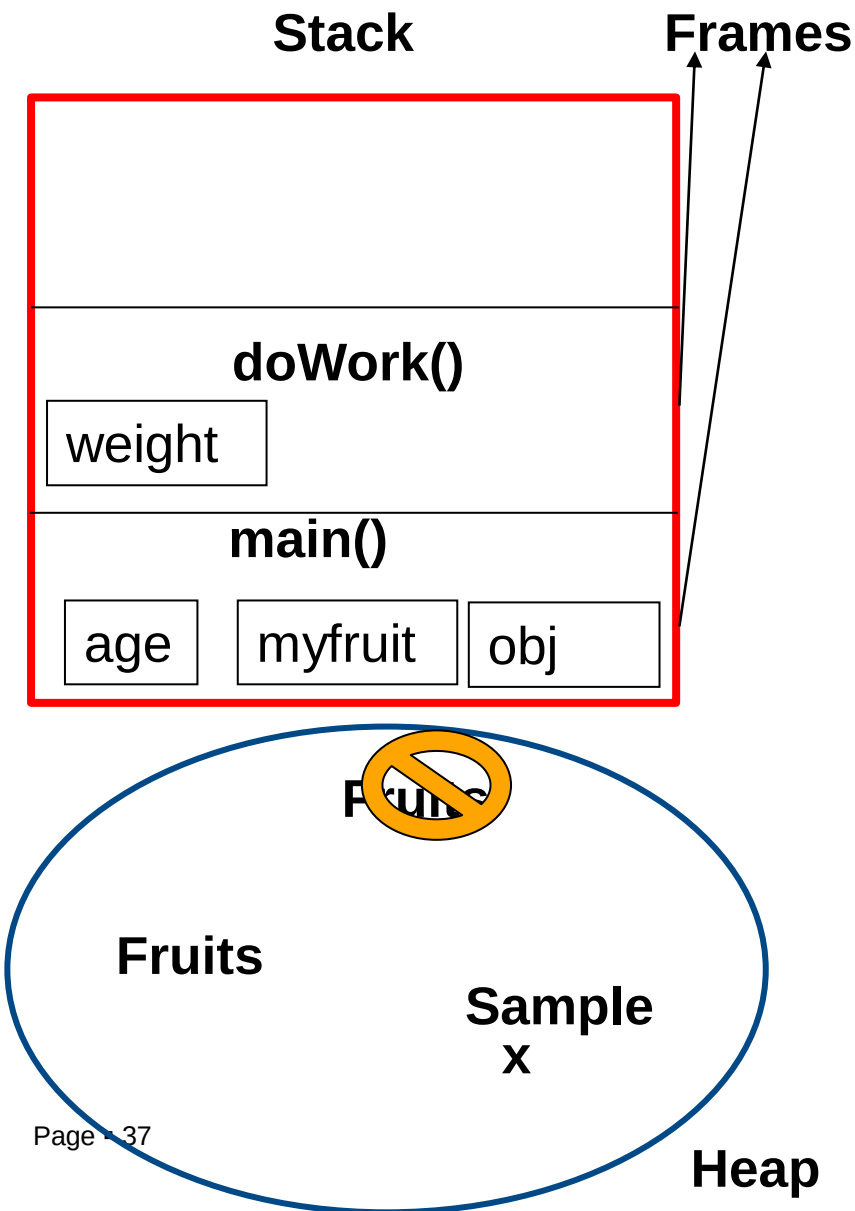
```
class Calculate
{
  static int x=5;
  static int cube()
  {
  return x*x*x;
  }
  public static void main(String args[])
  {
  int result=Calculate.cube();
  System.out.println(result);
  }
}
```

Static fields and methods

```
class Student
  {
   int rollno;//instance variable
   String name;
   static String college ="IT";
   Student(int r,String n)
 {
   rollno = r;
   name = n;
 System.out.println(rollno+" "+name+"
");
    }
static void display ()
{
   System.out.println(college);
 }
  }
```

```
public class TestStaticVariable1
{
 public static void main(String args[])
{
 Student s1 = new Student(111,"Aanya");
 Student s2 = new Student(222,"Arya");
  Student.college="PVC";
  Student.display();
  Student.display();
 }
}
```

Java Memory Management




```
class Sample
{
   int x;
   public static void main(String args[])
   {
   int age;
   age=10;
   age=11;

    doWork();

    Fruits myfruit;
    myfruit=new Fruits();
    fruit=new Fruits();
    Sample obj=new Sample();
    obj.x=10;
   }
}
```

Java Garbage Collection

- In java, garbage means unreferenced objects.
- **Garbage Collection** is process **of reclaiming the runtime unused memory automatically**. In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.
- **Advantages of Garbage Collection:**

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

finalize() method

- The **finalize()** method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:
- Syntax:

  protected void finalize(){}

- The Garbage collector of JVM collects only **those objects that are created by new keyword**. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

- The gc() method is used to invoke the **garbage collector** to perform **cleanup processing**. The gc() is found in System and Runtime classes.

  public static void gc(){}

- Garbage collection is performed by a **daemon thread called Garbage Collector(GC)**.
- This thread calls the **finalize() method** before object is garbage collected**.**
- **Neither finalization nor garbage collection is guaranteed.**

Example of garbage collection in java

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

```
object is garbage collected
object is garbage collected
```

Packages

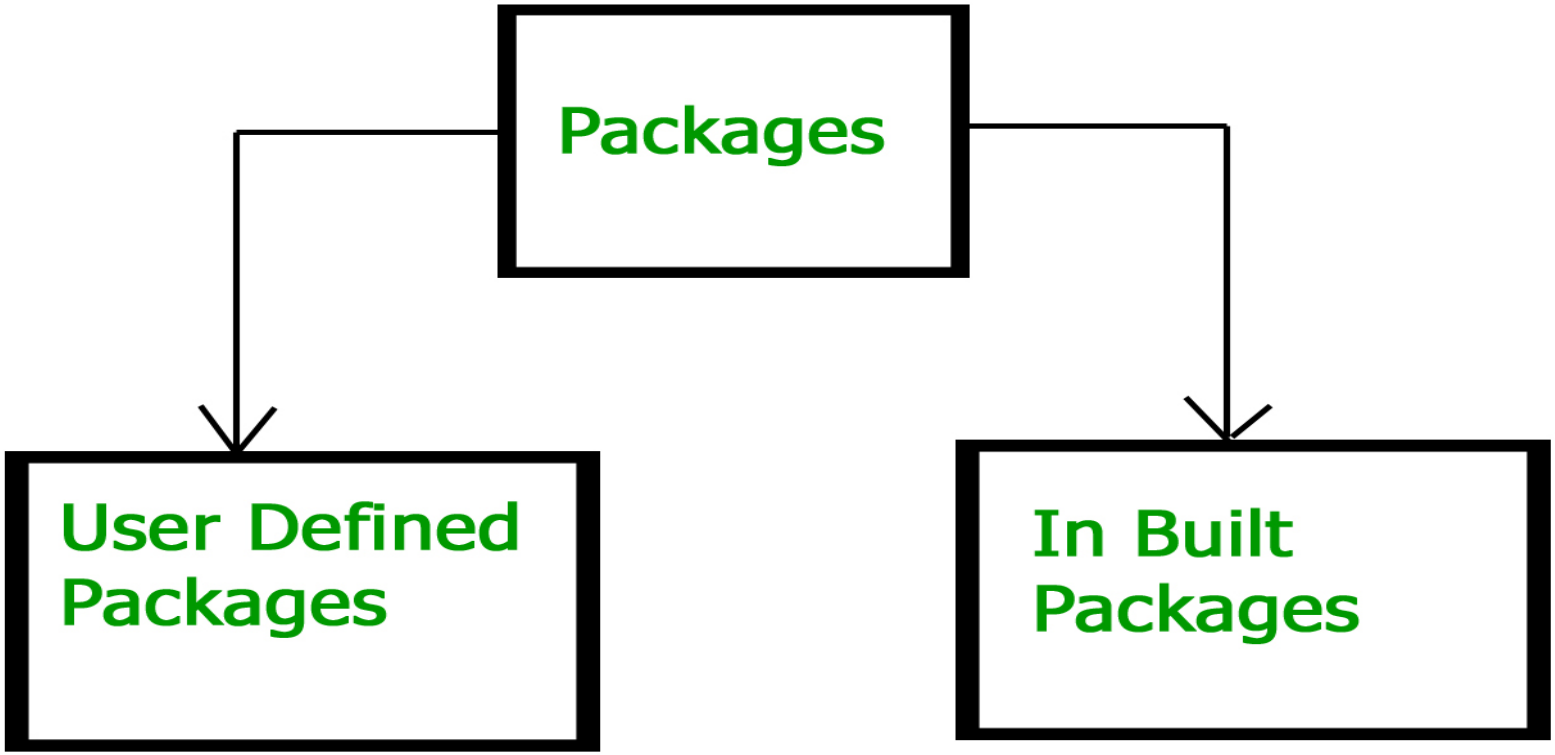
- A **package** as the name suggests is a pack(group) of classes, interfaces and other packages.
- In java we use packages to organize our classes and interfaces.
- We **have two types of packages** in Java**: built-in packages** and the packages we can create (also known as **user defined package**).
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- **Advantage of Java Package**

1) Java package is used to **categorize the classes and interfaces** so that they can be easily maintained.

2) Java package provides **access protection**.

3) Java package **removes naming collision.** For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee.

Packages

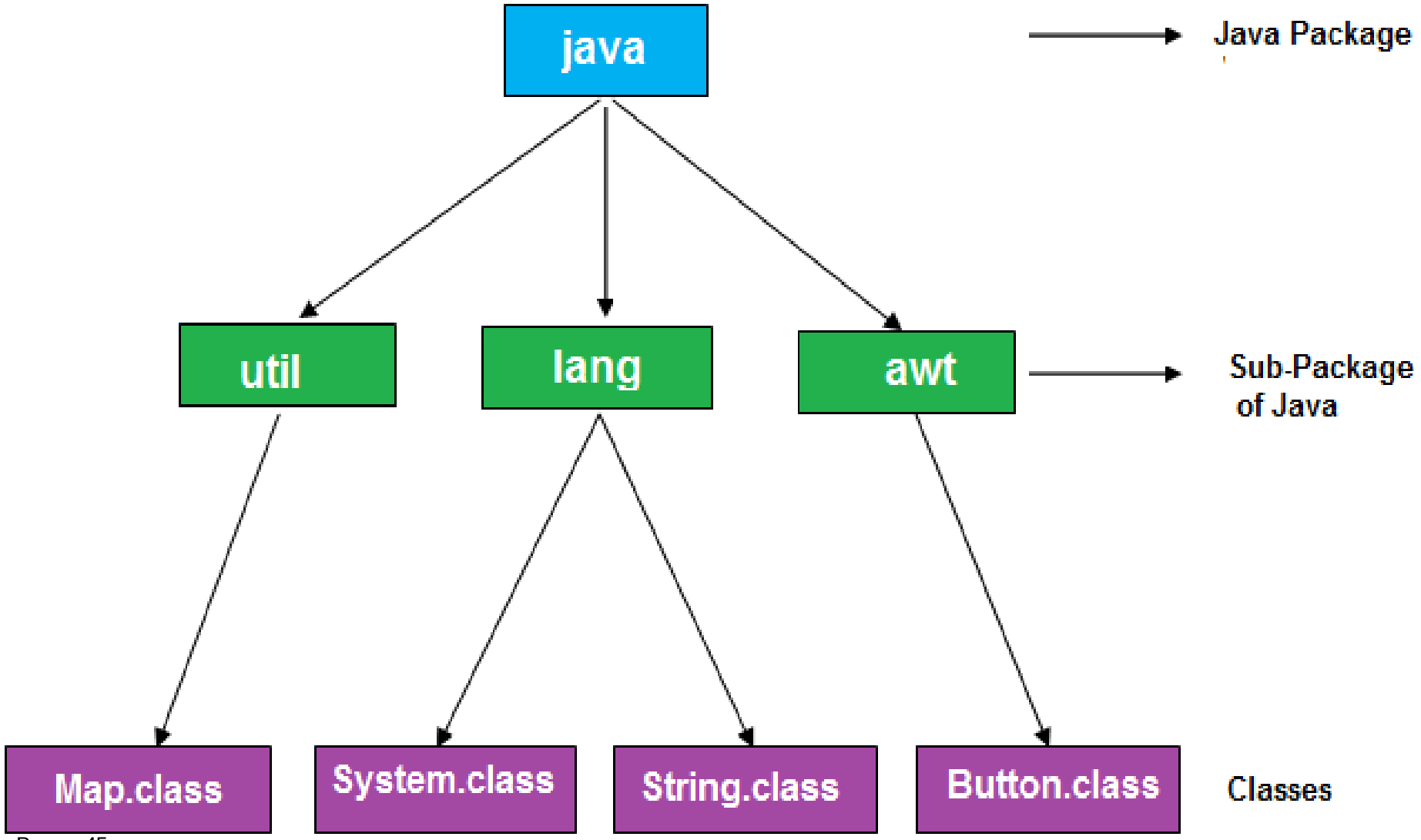


Packages

- Package names and directory structure are closely related.
- **Package naming conventions :**
- Packages are named in **reverse order of domain names**, i.e., org.abc.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

- **Adding a class to a Package :** We can add more classes to an created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.
- **Subpackages:** Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Packages


java
Java Package
util
lang
awt
Sub-Package
of Java
Map.class
System.class
String.class
Button.class
Classes

Creating Packages

```
//save as Simple.java
package mypack;
public class Simple
{
 public static void main(String args[])
  {
    System.out.println("Welcome to package");
   }
}
```

Creating Packages

➢ **How to compile java package**

If you are not using any IDE, you need to follow the syntax given below:

javac -d directory javafilename

For example, **javac -d . Simple.java**

➢ The -d switch specifies the destination where to put the generated class file.

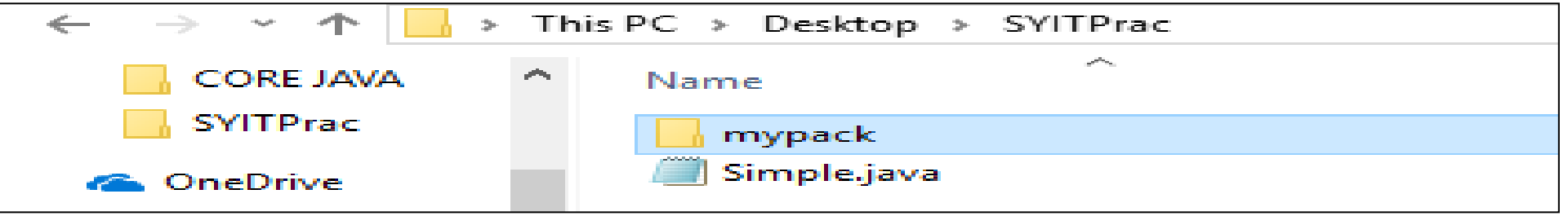
➢ You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

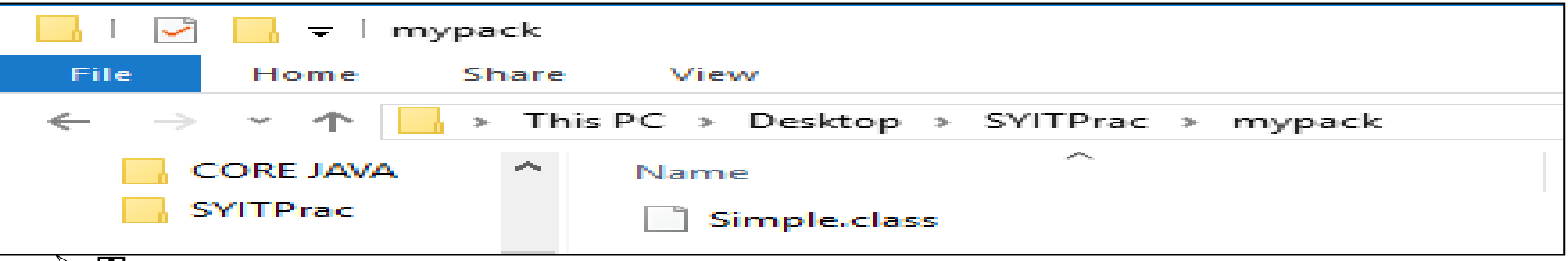
➢ **How to run java package program?**

- You need to use fully qualified name e.g. mypack.Simple, etc to run the class.
- **To Compile:** javac -d . Simple.java
- **To Run:** java mypack.Simple
- **Output:** Welcome to package
- The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

➢ **To compile:**

```
C:\Users\Virendra\Desktop\SYITPrac>javac -d . Simple.java
```





➢ **To run:**

```
C:\Users\Virendra\Desktop\SYITPrac>java mypack.Simple
Welcome to package
```

```
C:\Users\Virendra\Desktop\SYITPrac>javac -d D: Simple.java

C:\Users\Virendra\Desktop\SYITPrac>D:

D:\>java mypack.Simple
Welcome to package
```

Importing Package

- **There are three ways to access the package from outside the package.**
  - **import package.*;**
  - **import package.classname;**
  - **fully qualified name.**

1) **Using packagename.***

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Importing Packages

```
package pack;
public class Demo
{
  public void msg()
  {
   System.out.println("Hello");
  }
}
```

```
package mypack;
import pack.*;


class DemoRun
{
  public static void main(String args[]){
   Demo obj = new Demo();
   obj.msg();
  }
}
```

Importing Package

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

```
package pack;
public class Demo
{
  public void msg()
  {
   System.out.println("Hello");
  }
}
```

```
package mypack;
import pack.Demo;


class DemoRun
{
  public static void main(String args[]){
   Demo obj = new Demo();
   obj.msg();
  }
}
```

Importing Package

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Importing Packages

```
package pack;
public class Demo
{
  public void msg()
  {
   System.out.println("Hello");
  }
}
```

```
package mypack;
class DemoRun
{
  public static void main(String args[]){
   pack.Demo obj = new pack.Demo();
   obj.msg();
  }
}
```