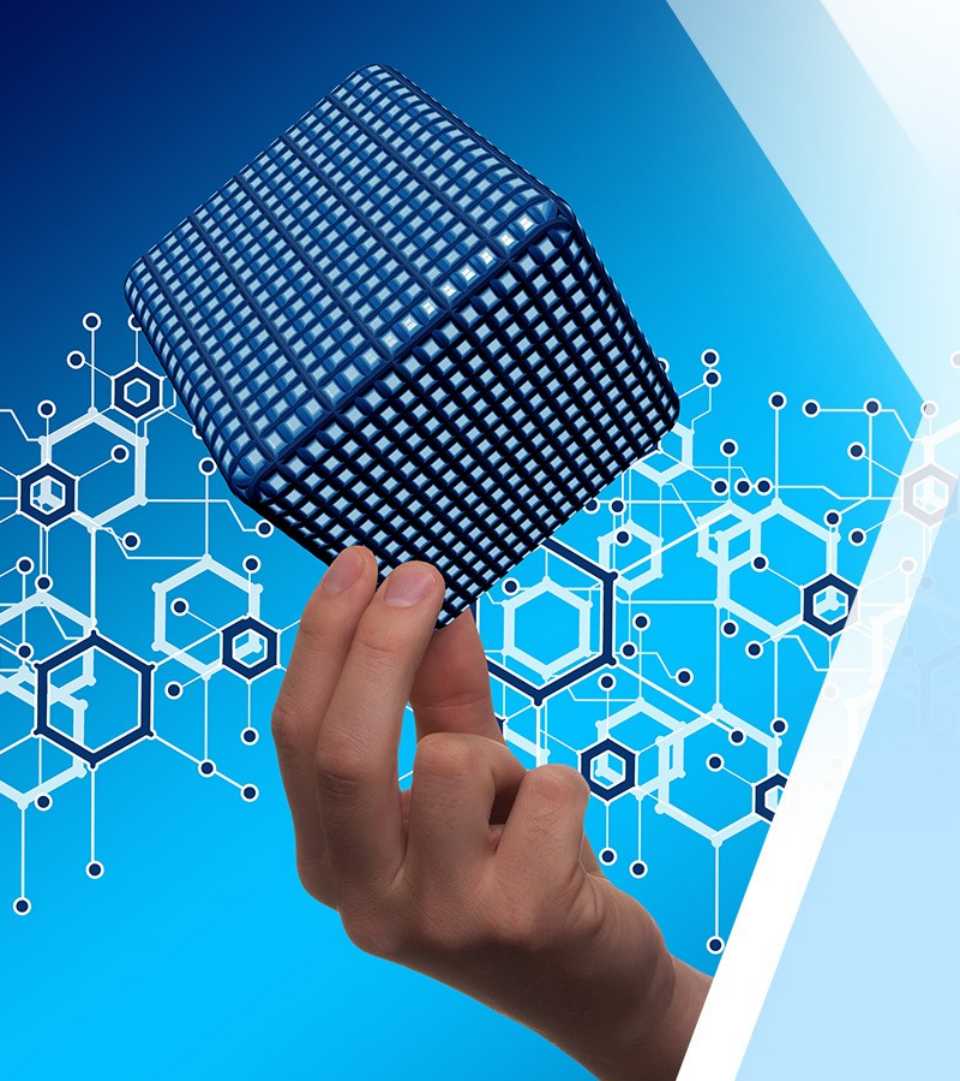
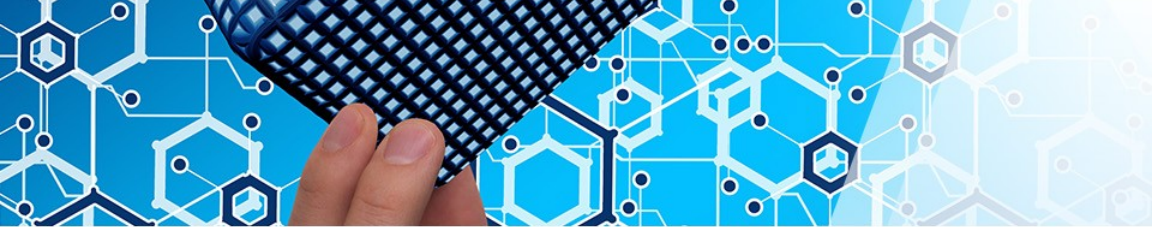




Chapter 4

Objects and Classes



Defining a Class

- A class is a template/blue print/ design that describes the behaviors/states that object of its type support.
- A class can have any number of methods to access the value of various kinds of methods.
- Class definition consists
 - **Instance /class variable** as a data members
 - **Methods** as member functions (Actuator and mutator)
- You could have several class definition in a program

Class Example

```
public class Dog
```

```
{
```

```
    String breed;
```

```
    int age
```

```
    String color;
```

```
    void barking() { //body ..... }
```

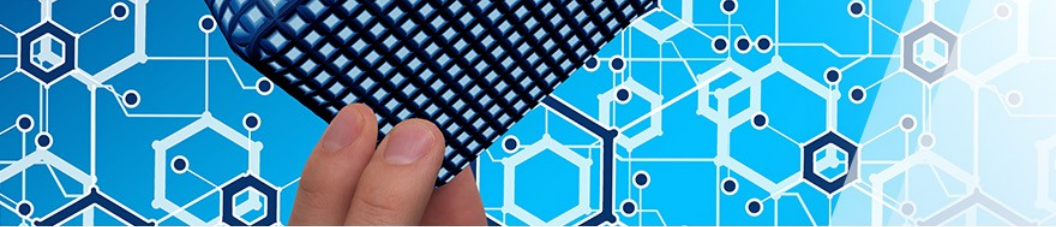
```
    void hungry() { //body ..... }
```

```
    void sleeping() { //body ..... }
```

```
}
```

- This class has 3 methods and 3 instance variables.
- Methods like:- **barking()**, **hungry()**, **sleeping()**
- States like:- **breed**, **age**, **color**.





Creating Objects

- A class provides the blueprints for objects.
- So basically an object is created from a class/**instance of a class**.
- In java, the **new keyword** is used to create new objects.

There are three steps when creating an object from a class

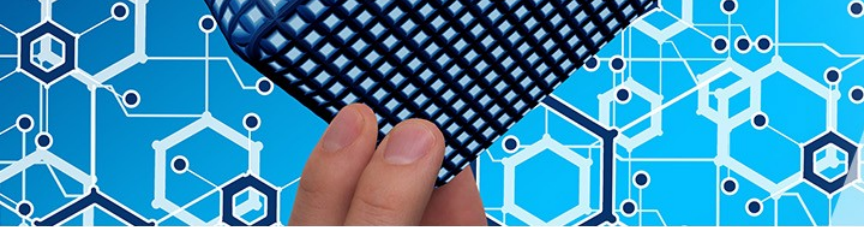
1. **Declaration**- a variable declaration with a variable name of an object type
2. **Instantiation**- the 'new' key word is used to create the object.
3. **Initialization**- the 'new' key word is followed by a **call to a constructor**. This call initializes the new object.



Example on creating object

```
public class Puppy{  
  
    public Puppy(String name){  
        // This constructor has one parameter, name.  
        System.out.println("Passed Name is :" + name );  
    }  
  
    public static void main(String []args){  
        // Following statement would create an object myPuppy  
        Puppy myPuppy = new Puppy( "tommy" );  
    }  
}
```

- If we run this program the output will be
- Passed Name is : tommy



Instantiating and using objects

- Instance variables and methods are **accessed via created objects**.
- To access an instance variable the fully qualified path should be .

```
/* First create an object */  
ObjectReference = new Constructor();  
  
/* Now call a variable as follows */  
ObjectReference.variableName;  
  
/* Now you can call a class method as follows */  
ObjectReference.MethodName();
```

```
public class Puppy{
```

```
    int puppyAge;
```

```
    public Puppy(String name){
```

```
        // This constructor has one parameter, name.
```

```
        System.out.println("Name chosen is :" + name );
```

```
    }
```

```
    public void setAge( int age ){
```

```
        puppyAge = age;
```

```
    }
```

```
    public int getAge( ){
```

```
        System.out.println("Puppy's age is :" + puppyAge );
```

```
        return puppyAge;
```

```
    }
```

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

```
        /* Object creation */
```

```
        Puppy myPuppy = new Puppy( "tommy" );
```

```
        /* Call class method to set puppy's age */
```

```
        myPuppy.setAge( 2 );
```

```
        /* Call another class method to get puppy's age */
```

```
        myPuppy.getAge( );
```

```
        /* You can access instance variable as follows as well */
```

```
        System.out.println("Variable Value :" + myPuppy.puppyAge );
```

```
    }
```

```
}
```

Accessing instance variables, methods

```
C:\MyJavaApplications\Chapter 4>javac Puppy1.java
```

```
C:\MyJavaApplications\Chapter 4>java Puppy1
```

```
Name chosen is :tommy
```

```
Puppy's age is :2
```

```
Variable Value :2
```

- A class can contain any of the following variable types

- 1. Local variables**
- 2. Instance variables**
- 3. Class variables**

Local Variable Example

```
public class Test{  
    public void pupAge(){  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]){  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

2. Instance Variables Example

```
public class Employee{
    // this instance variable is visible for any child class.
    public String name;

    // salary variable is visible in Employee class only.
    private double salary;

    // The name variable is assigned in the constructor.
    public Employee (String empName){
        name = empName;
    }

    // The salary variable is assigned a value.
    public void setSalary(double empSal){
        salary = empSal;
    }

    // This method prints the employee details.
    public void printEmp(){
        System.out.println("name : " + name );
        System.out.println("salary :" + salary);
    }

    public static void main(String args[]){
        Employee empOne = new Employee("Ransika");
        empOne.setSalary(1000);
        empOne.printEmp();
    }
}
```

3. Class Variable example

```
public class Circle {
    public static float pi = 3.14159265F;

    public float area(float r) {
        return pi * r * r;
    }
}

public class CircleTest {
    public static void main(String [] args){
        Circle c=new Circle();
        float area=c.area(3.0f);
        System.out.println("Area="+area);

    }}
}
```

```
C:\MyJavaApplications\Chapter 4>javac CircleTest.java
```

```
C:\MyJavaApplications\Chapter 4>java CircleTest
Area=28.274334
```



Example: Class/static variables

```
public class Employee{  
    // salary variable is a private static variable  
    private static double salary;  
  
    // DEPARTMENT is a constant  
    public static final String DEPARTMENT = "Development ";  
  
    public static void main(String args[]){  
        salary = 1000;  
        System.out.println(DEPARTMENT+"average salary:"+salary);  
    }  
}
```



Class Member Visibility

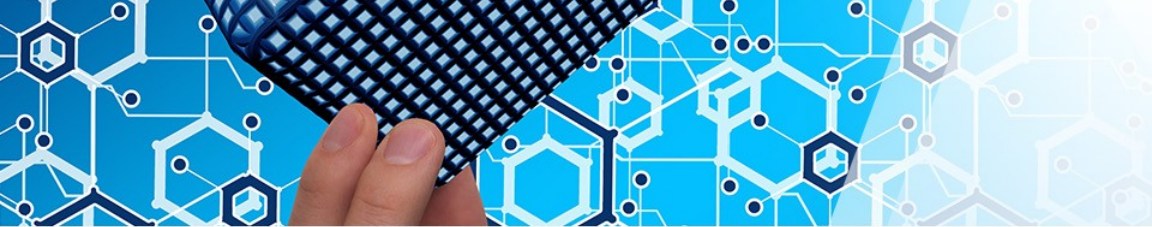
- **Modifiers**

- Are **keywords** that you add to those definitions to **change their meanings**.
- **Modifiers are prefixes** that can be applied in various combinations to the **methods and variables within a class and , some, to the class itself**.
- There are two types of access modifiers
 - Java Access modifiers
 - Non Access Modifiers



Access Control Modifiers

- Java provides a number of access modifiers **to set access levels for classes, variables, methods and constructors. The four access levels are:**
 - Visible to the package, the default. No modifiers are needed.
 - Visible to the class only (**private**).
 - Visible to the world (**public**).
 - Visible to the package and all subclasses (**protected**).



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**.

- Any method or variable is visible to the class in which it is defined,
- but what if you want to make method/variable visible to *all* the classes outside this class?
- The answer is obvious: simply declare the method or variable to have public access.
- A variable or method with public access has the **widest possible visibility**. Anyone can see it. Anyone can use it.



Public modifier example

```
public class APublicClass {  
    public int      aPublicInt;  
    public String   aPublicString;  
  
    public float    aPublicMethod() {  
        . . .  
    }  
}
```

- It has no **precise/specific name**
- **No access modifier specified** in a declaration.
- it will be possible to say **package explicitly**, but for now it is simply the **default protection** when nothing has been specified as a name.



Default access modifier -package

```
public class ALessPublicClass {  
    int    aPackageInt    = 2;  
    String aPackageString = "a 1 and a ";  
  
    float  aPackageMethod() {    // no access modifier means "package"  
        . . .  
    }  
}  
  
public class AClassInTheSamePackage {  
    public void testUse() {  
        ALessPublicClass aLPC = new ALessPublicClass();  
  
        System.out.println(aLPC.aPackageString + aLPC.aPackageInt);  
        aLPC.aPackageMethod();    // all of these are A.O.K.  
    }  
}
```

- The third relationship is between a class and its present and future subclasses.
- These **subclasses are much closer to a parent** class than to any other “outside” classes for the following reasons:
 - Subclasses are usually more intimately aware of the internals of a parent class
 - Subclasses are often written by you or by someone to whom you’ve given your source code.
 - Subclasses frequently need to modify or enhance the representation of the data within a parent class.

Protected class

```
public class AProtectedClass {
    protected int    aProtectedInt    = 4;
    protected String aProtectedString = "and a 3 and a ";

    protected float  aProtectedMethod() {
        . . .
    }
}

public class AProtectedClassSubclass extends AProtectedClass {
    public void testUse() {
        AProtectedClass aPC = new AProtectedClass();

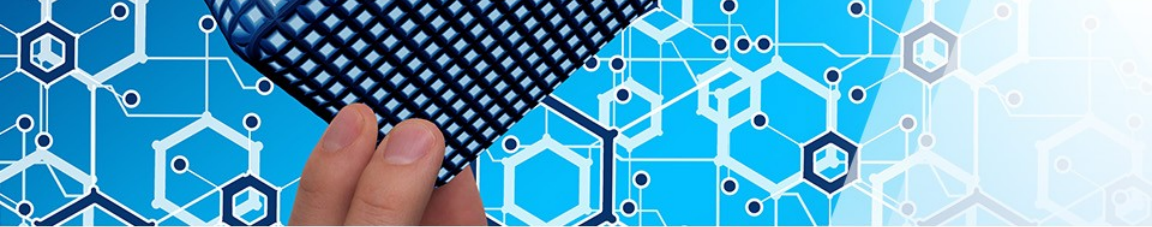
        System.out.println(aPC.aProtectedString + aPC.aProtectedInt);
        aPC.aProtectedMethod();           // all of these are A.O.K.
    }
}

public class AnyClassInTheSamePackage {
    public void testUse() {
        AProtectedClass aPC = new AProtectedClass();

        System.out.println(aPC.aProtectedString + aPC.aProtectedInt);
        aPC.aProtectedMethod();           // NONE of these are legal
    }
}
```



- Even though **AnyClassInTheSamePackage** is in the same package as **AProtectedClass**, it is not a subclass of it (it's a subclass of Object).
- Only **subclasses** are allowed to see, and use, **protected** variables and methods.



Private

- Private is the **most narrowly visible**, highest level of protection that you can **get**.
- It is the **diametric opposite of public**. private methods and variables cannot be seen by any class other than the one in which they are defined .
- Any **private data, internal state, or representations unique to your implementation** anything that shouldn't be directly shared with subclasses is private.



- Remember that an **object's primary job is to encapsulate** its data to hide it from the world's sight and limit its manipulation.
- The best way to do that is to make as much data as private as possible

```
public class APrivateClass {  
    private int      aPrivateInt;  
    private String   aPrivateString;  
  
    private float    aPrivateMethod() {  
        . . .  
    }  
}
```



- By convention, unless an **instance variable is constant** , it should almost certainly be **private**.



Access Modifiers Summary

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



Accessor(**get**) Mutator(**set**) Methods

- If instance variables are private, how do you give access to them to the outside world?
- The answer is to write “accessor” and “mutator” methods:

```
public class  ACorrectClass {  
    private   String  aUsefulString;  
  
    public    String  aUsefulString() {           // "get" the value  
        return aUsefulString;  
    }  
  
    protected void    aUsefulString(String s) {   // "set" the value  
        aUsefulString = s;  
    }  
}
```



- **a class's private fields** can be manipulated only by its methods.
- Classes often provide public methods to allow clients of the class to **set** (i.e., assign values to) or **get** (i.e., obtain the values of) private instance variables.
- **Set** methods are also commonly called **mutator methods**, because they typically change an object's state—i.e., modify the values of instance variables.
- **Get** methods are also commonly called **accessor methods** or **query methods**.

```
class PoorDog
{
    private int size;
    private String name;
    public void setSize(int sizeLocal){
        size = sizeLocal;
    }
    public void setName(String nameLocal){
        name = nameLocal;
    }
    public int getSize(){
        return size;
    }
    public String getName()
    {
        return name;
    }
}
```



See [PoorDog.java](#) & [PoorDogTestDrive.java](#)

```
public class PoorDogTest {  
  
    public static void main(String[] args) {  
        PoorDog pdog = new PoorDog();  
        pdog.setName("Tommy");  
        pdog.setSize(30);  
        System.out.println("Dog's Name:" + pdog.getName());  
        System.out.println("Dog's Size:" + pdog.getSize());  
    }  
}
```

```
class Student {  
    String ID; String fName; String lName;  
    String gender;  
    int age;  
    void setId(String idn){ID= idn;}  
    void setFirstName(String fname){fName = fname;}  
    void setLastName(String lname){ lName = lname;}  
    void setAge(int a){age = a;}  
    void setGender(String sex){gender = sex;}  
    String getfName() {return fName;}  
    String getId() {return ID;}  
    String getlName() {return lName;}  
    int getAge() {return age;}  
    String getGender() {return gender;}  
}
```

Application with two classes

```
import java.util.Scanner;
public class TestStudent
{
    public static void main(String []arg)
    {
        Student s1 = new Student();
        Scanner input = new Scanner(System.in);
        System.out.print("ID:");
        s1.setId(input.next());
        System.out.print("FName:");
        s1.setFirstName(input.next());
        System.out.print("LName:");
        s1.setLastName(input.next());
        System.out.print("Age:");
        s1.setAge(input.nextInt());
        System.out.print("Sex:");
        s1.setGender(input.next());
        System.out.println("ID\tFName\tLname\tAge\tSex");
        System.out.println(s1.getId()+"\t"+s1.getfName()+"\t"+ s1.getlName()+"\t"+s1.getAge()+"\t"+ s1.getGender());
    }
}
```



End of Chapter IV



CHAPTER FIVE