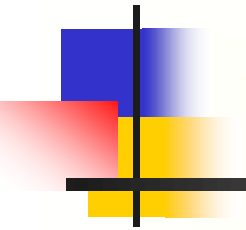


SWENG3101

Object Oriented Programming

Software Engineering Department

AASTU

Chapter 3

Classes and Objects

Objectives

At the end of this chapter you will:

- describe objects and classes
- use classes to model objects
- access an object’s data and methods
- distinguish between instance and static variables and methods
- create objects using constructors
- use the keyword **this** to refer to the calling object itself

Content

- Classes and Objects
- Access Control (private, protected, public)
- Attributes and methods
- Constructors

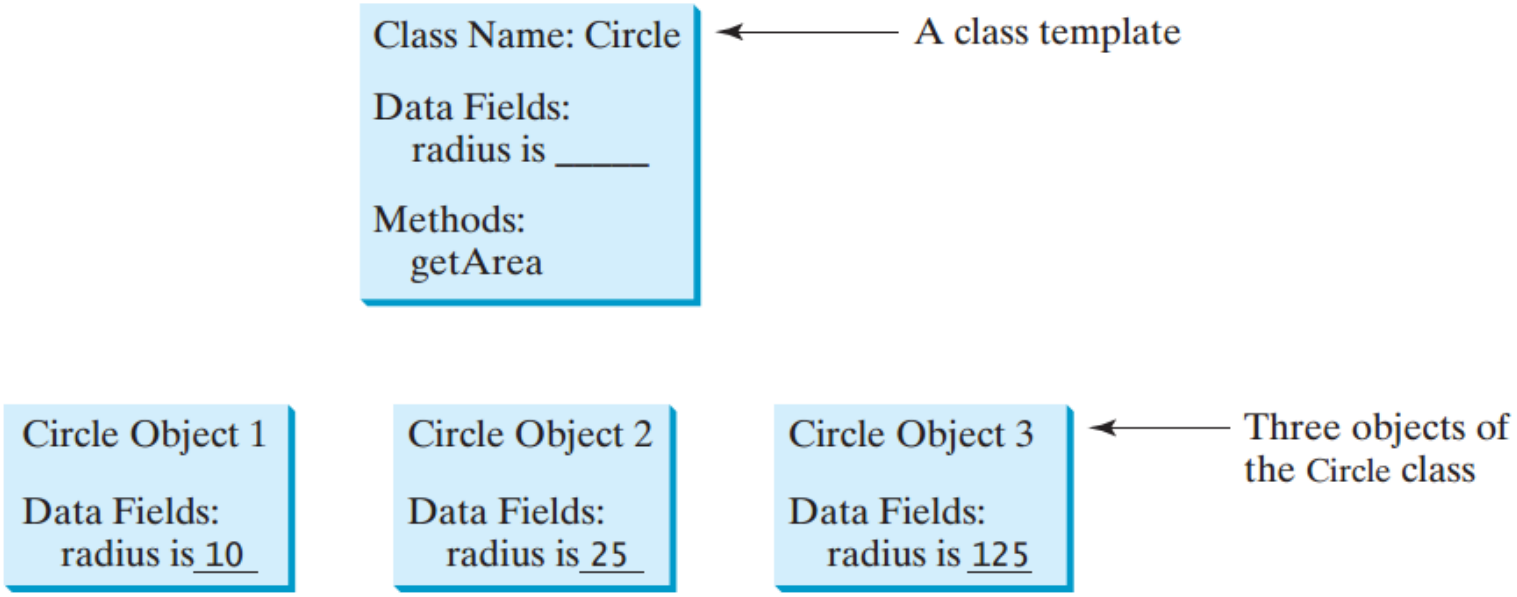
Classes and Objects

- OOP involves programming using objects.
- The two most important concepts in object-oriented programming are the class and the object.
- **Object**
  - Is an entity, both tangible and intangible, in the real world that can be distinctly identified.
    - E.g. Student, Room, Account
  - An object is comprised of data and operations that manipulate these data.
  - E.g.1 For a **Student**
    - **Data(property/attribute):** name, gender, birth date, home address, phone number, age, …
    - **And operations** for assigning and changing these data values.
  - E.g. 2 **dog**,
    - data - name, breed, color, …
    - behavior(operation) - barking, wagging, running.

Classes and Objects(cont'd)

- **Class**
  - A class is a template or blue print that defines the properties and behaviors for objects
  - Java class uses variables to define data fields and methods to define actions
  - Each object can have different data, but all objects of a class have the same types of data and methods
  - A class is a kind of mold or template that dictates what objects **can** and **cannot do**.
  - An object is called an instance of a class.
  - Once a class is defined, we can create as many instances of the class as a program requires.
  - Creating an instance is referred to as instantiation.

Example: Class and object



Declaring Class

- Class Declaration syntax

```
class ClassName {
    // declare instance variables
    type var1;
    type var2;
    // ...
    type varN;
    // declare methods
    type method1(parameters) {
       // body of method
    }
    type method2(parameters) {
       // body of method
    }
    // ...
    type methodN(parameters) {
        // body of method
    }
}
```

Creating Objects

- Create an object from a class.
- Syntax:
  - *ClassName objectname; //object declaration*
  - *objectname=new ClassName (parameters); // object creation*

    Or

    ClassName objectname=new ClassName (parameters);
  - E.g. If we have the following class definition for customer:

```
class Customer{
        //variables
         public void printCust(){
                //method definition
        }
}
```

    we can create “cust” objcte as follows:
    - Customer cust = new Customer();

Accessing data members

- To access data members, and methods of the class, we use dot(.) operator.
  - *Objectname.data member //to access the data member of the class*
  - *Objectname.method* // to access the method of the class
    - E.g. cust.printCust();

Example 1

```
public class GradeBook  {
    public void displayMessage() {
        System.out.println ("Welcome to the Grade Book!");
    }
}
```

- *A second class GradeBookTest uses and executes the method declared in Gradebook class*

```
public class GradeBookTest{
    public static void main (String args[] ) {
        GradeBook myGradeBook = new GradeBook();
        myGradeBook.displayMessage();
    }
}
```

Example 2: *defining a class for circle object*

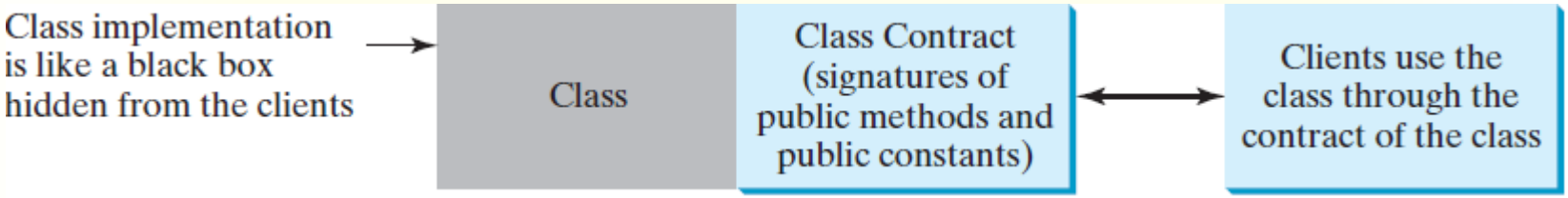
```
public class Circle {
   Double pi=3.14;
   Double rad;
   Double area(){
     return (pi*rad*rad) ;
    }
    Double circumf (){
         return (2*pi*rad);
     }
}
```

Cont'd

```
public class TestCircle {
    public static void main(String args[]){
        Circle c1=new Circle();
        c1.rad=3.0;
        System.out.println ("area of the circle is:"+c1.area());
       System.out.println ("circumference of the circle is:"+c1.circumf ());
    }
}
```

Class Abstraction and Encapsulation

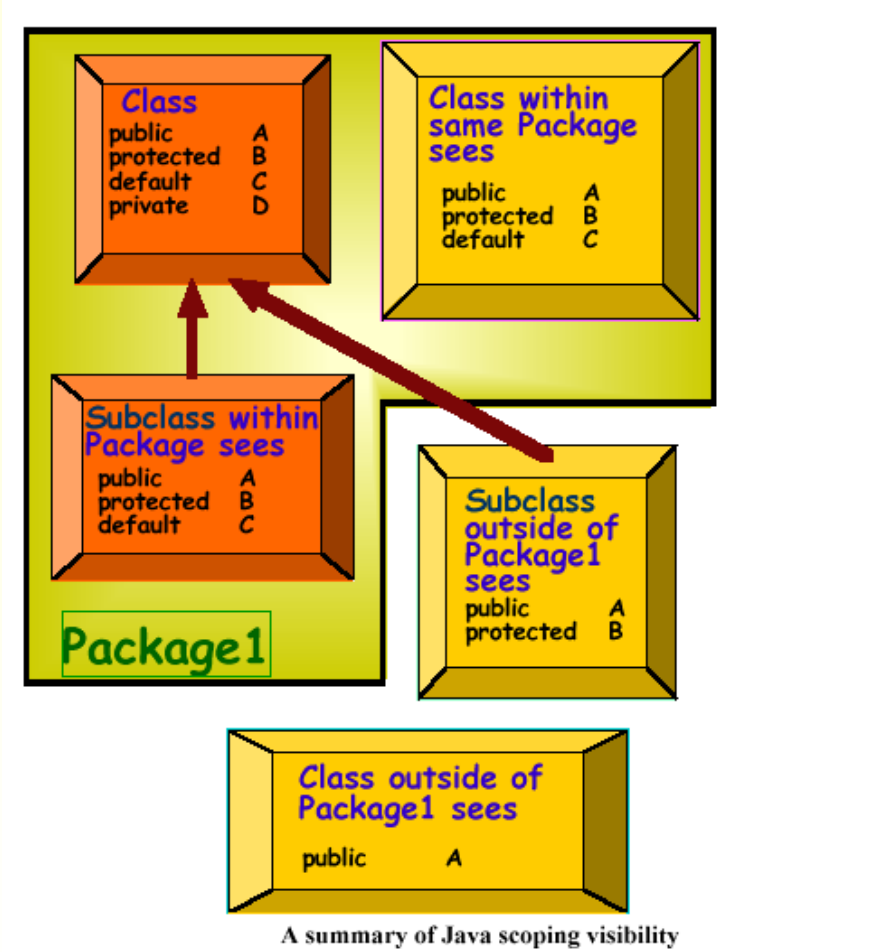
- Class **abstraction** is the separation of class implementation from the use of a class.
  - This is also known as *information hiding* or *encapsulation*
- **Encapsulation** means that the data and the actions are combined into a single item(in a class object) and that the details of the implementation are hidden



Access control

- Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors.
- The access modifiers are: private, protected, public and default
- Access to class members (in class C in package P)
  - Public: accessible anywhere C is accessible
  - Protected: accessible in P and to any of C's subclasses
  - Private: only accessible within class C
  - If you don't use any modifier: only accessible in P (the default)

Access control(cont'd)



A summary of Java scoping visibility

Example

```
// public vs private access.
class MyClass {
   private int alpha; // private access
   public int beta; // public access
   public int gamma; // public access
   //Methods to access alpha. It is OK for a member of a class to
    //access a private member of the same class.
   void setAlpha(int a) {
      alpha = a;
   }
   int getAlpha() {
      return alpha;
   }

}
```

Example(cont'd)

```
class AccessDemo {
   public static void main(String args[]) {
      MyClass ob = new MyClass();
      //Access to alpha is allowed only through its accessor method.
      ob.setAlpha(-99);
      System.out.println("ob.alpha is " + ob.getAlpha());
      // You cannot access alpha like this:
      // ob.alpha = 10; // Wrong! alpha is private!
      // These are OK because beta and gamma are public.
      ob.beta = 88;
      ob.gamma = 99;
      System.out.println("ob.beta is " + ob.beta);
      System.out.println("ob.gamma is " + ob.gamma);

   }
}
```

Activity

- Based on the following sample code and information, identify the correct statements (from a-d)

```
public class ClassA{
    int a;
     public int b;
     private int c;
     protected int d;
      ….
}
```

- ClassA & ClassB are defined under package1, Class C is defined under Package2 and ClassD, which is sub class of ClassB, is defined under Package3
  - (a) ClassB{ClassA ca = new ClassA; ca.a; }
  - (b) ClassC{ClassA cb = new ClassA; cb.b;}
  - (c) ClassD{ClassA cc = new ClassA; cc.c ;}
  - (d) ClassD{ClassA cd = new ClassA; cd.d }

Local, instance and class (static) variables

- **Local variables:**
  - Variables defined inside methods, constructors or blocks are called local variables.
  - The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  - Local variables are **visible only within the declared method, constructor** or **block.**
  - There is no default value for local variables so local variables should be declared and an **initial value should be assigned** before the first use.

Example: Local Variables

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

Output

Puppy age is : 7

Example(cont'd)

- Following example uses *age* without initializing it, so it would give an error at the time of compilation.

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

Error: variable age might not have been initialized.

Instance Variables

- Are variables within a class but outside any method.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables can be declared in class level **before** or **after use**.
- Instance variables can be **accessed directly by calling the variable name inside the class**. However within static methods and different class should be called using the **fully qualified name**.
  - E.g. **ObjectName.variable**

Instance Variables(cont'd)

- Access modifiers can be given for instance variables.
- Instance variables **have default values**. For numbers the default value is 0, for Booleans it is false and for object references it is null.
- Values can be assigned during the declaration or within the constructor.

Example1

```
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 a value.
   public void setName(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);
   }
```

Example1(cont'd)

```
public static void main(String args[]){
      Employee empOne = new Employee();
      Employee empTwo = new Employee();
      empOne.setName("John");
      empTwo.setName("Max");
      empOne.setSalary(1000);
      empTwo.setSalary(2000);
      empOne.printEmp();
      empTwo.printEmp();
   }
}
```

Output:

name : John
salary :1000.0
name : Max
salary :2000.0

Example2

```
public class Bank {
     int balance;
  void setbalance(int bal){
       balance=bal;
   }
   int deposite( int dep){
        balance=balance + dep;
       return balance;
   }
   int withdraw(int withd){
       balance=balance - withd;
       return balance;
   }
}
```

Example2(cont'd)

```
public class Runbank {
    public static void main(String arg[]){
        Bank cust1= new Bank();
        cust1.setbalance(1000);
        System.out.println("customer 1 balance after withdrawal "+
    cust1.withdraw(300));
        System.out.println(" Customer 1 balance after depositing  "+
    cust1.deposite(3000));
        Bank cust2= new Bank();
        cust2.setbalance(5000);
        System.out.println(" customer 2 balance after withdrawal  "+
    cust2.withdraw(500));
        System.out.print(" customer 2 balance after depositing  "+
    cust2.deposite(3000));
    }
  }
```

Example 3: Predict output

```
public class Foo {
   private boolean x;
   public static void main(String[] args) {
      Foo foo = new Foo();
      System.out.println(foo.x);
   }
}
```

Class/static variables

- Class variables also known as static variables are declared with the ***static*** keyword in a class, but outside a method, constructor or a block.
- Every object has its own copy of all the ***instance variables*** of the class. But, there would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as **constants**.
- Static variables are created when the program starts and destroyed when the program stops.
- Default values are same as instance variables.

Example1

```
public class Employee2{
   // salary  variable is a private static variable
   private static double salary;

   // DEPARTMENT is a constant
   public static String DEPARTMENT = "Development ";

   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT + "average salary:" +
    salary);
   }
}
```

Example 2

```
public class Employee3 {
   // this instance variable is visible for any child class.
   public String name;
   public  static double initialsalary;
   // The name variable is assigned in the constructor.
   public Employee3 (String empName){
      name = empName;
   }
   // The salary variable is assigned a value.
   public void setSalary(double empSal){
      initialsalary = empSal;
   }
   // This method prints the employee details.
   public void printEmp(){
      System.out.println("name : " + name );
      System.out.println("initial salary :" + initialsalary);
   }
}
```

Example2(cont'd)

```
public class Employee3Test{
   public static void main(String args[]){
      Employee3 empOne = new Employee3("Ahmed");
      empOne.setSalary(1000);
      empOne.printEmp();
      Employee3 emptwo = new Employee3("Kedir");
    emptwo.printEmp();
  }
}
```

Example 3: Predict output

```
class Counter{
    int count=0;// what if it is static int count=0;
    public void printCounter(){
        count++;
        System.out.println(count);
    }
    public static void main(String args[]){
        Counter c1=new Counter();
        c1.printCounter();
        Counter c2=new Counter();
        c2.printCounter();
        Counter c3=new Counter();
        c3.printCounter();
    }
}
```

Methods

- Method describe behavior of an object.
- A method is a collection of statements that are group together to perform an operation.
- Java methods are like C/C++ functions.
- General case:

  [modifier] returnType methodName (*arg1*, *arg2*, … *argN*) {

  methodBody

  }

- The body of a method that returns a value must contain at least one return statement
  - Form: return *Expression;*
- Example:

```
public String getFirstName() {
    return firstName;
}
```

Instance and Static Methods

- Methods are defined in a class, invoked using the class object.
- Normally a class member must be accessed only in conjunction with an object of its class. There will be times when you will want to define a class member that will be used independently of any object of that class.
- It is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**.
- When a member is declared static, it **can be accessed before any objects of its class are created**, and without reference to any object.
- The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance and Static Methods(cont'd)

Class methods	Instance Methods
Static methods are declared by using static keyword.	instance methods are declared without static keyword
All objects share the single copy of static method.	All objects have their own copy of instance method.
Static method does not depend on the single object because it belongs to a class	Instance method depends on the object for which it is available.
Static method can be invoked without creating an object, by using class name. **ClassName.method( );**	Instance method cannot be invoked without creating an object. ObjectName.method( );
Static methods cannot call non-static methods.	Non-static methods can call static methods
Static methods cannot access non-static variables.	Non-static methods can access static variables
Static methods cannot refer to **this** or **super**.	Instance methods can refer to **this** and **super**

Example 1

- Suppose that the class Foo is defined in (a). Let f be an instance of Foo. Which of the statements in (b) are correct?

- (a)

```
public class Foo {
   int i;
   static String s;
   void imethod() {
   }
   static void smethod() {
   }
}
```

- (b)

```
System.out.println(f.i);
System.out.println(f.s);
f.imethod();
f.smethod();
System.out.println(Foo.i);
System.out.println(Foo.s);
Foo.imethod();
Foo.smethod();
```

Example 2

```
public class HelloName {
    String name;
    public static void helloName(){
        String name;
        Scanner input= new Scanner(System.in);
        name = input.nextLine();
        System.out.println("Hello" + name);
    }
    public static void main(String args[]){
      System.out.print("What is your name? ");
      helloName();
   }
}
```

Activity

- Write a program that count all words in a string.

Declaring a method with a parameter

- A method can require one or more parameters that **represent additional information** it needs to perform its task.
- A method call supplies values—called arguments—for each of the method's parameters.
- For example, to make a deposit into a bank account, a deposit method specifies a **parameter that represents the deposit amount.**

Example

```
public class GradeBook2{
    public void displayMessage(String courseName){
       System.out.print( "Welcome to the grade book for"+ courseName );
     }
}
import java.util.*;
public class GradeBook2Test {
    public static void main( String[] args){
      Scanner input = new Scanner( System.in );
      GradeBook2 myGradeBook = new GradeBook2();
       System.out.print( "enter the course name” );
       String nameOfCourse = input.nextLine();
       myGradeBook.displayMessage( nameOfCourse );
    }
 }
```

Constructors

- **Constructor in java** is a **special type of method** that is used to create or construct instances of the class.
- Called when keyword ***new*** is followed by the class name and parentheses
- Their name is the same as the class name
- It looks like a method, however it is not a method. Methods have return type but constructors **don't have any return type**(even void).
- Format: *public ClassName(para){…}*
- A constructor with no parameters is referred to as a *no-arg constructor*.

Default Constructor

- A class may be declared without constructors.
- In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called *a* default constructor, is provided **automatically** only ***if no constructors are explicitly declared*** *in the class*.
- Format: *public ClassName(){…}*

Constructor Example

```
public class EmployeeConstructor{
 // 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 EmployeeConstructor (String empName){
     name = empName;
 }
 // The salary variable is assigned a value.
 public void setSalary(double empSal){
    salary = empSal;
 }
```

Constructor example(cont'd)

```
// 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[]){
EmployeeConstructor empOne = new EmployeeConstructor("James");
empOne.setSalary(1000);
empOne.printEmp();
}
}
```

Example 2

```
class Account {
    // Data Members
    private double balance;
    //Constructor
    public Account(double startingBalance) {
       balance = startingBalance;
    }
    //Adds the passed amount to the balance
    public void depos(double amt) {
       balance = balance + amt;
    }
    //Returns the current balance of this account
    public double getCurrentBalance( ) {
       return balance;
    }
}
```

Example2(cont'd)

```
import java.util.Scanner;
public class AccountTest{
   public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        Double bal, amt;
        Account acct = new Account(200);
        System.out.println("Please enter amount of deposit");
        amt= in.nextDouble();
        acct.depost (amt);
        System.out.println("The current balance is: " +
            acct.getCurrentBalance());

   }
}
```

Activity

- Which of the following constructors are invalid?

1. public int ClassA(int one) {

...

}

2. public ClassB(int one, int two) {

...

}

3. void ClassC( ) {

...

}

Multiple constructors

- A class can have multiple constructors, as long as their signature (the parameters they take) are not the same.
- You can define as many constructors as you need.

Example

```
public class CircleConstructor {
    public double r; //instance variable
      // Constructors
    public CircleConstructor(double radius) {
          r = radius;
    }
    public CircleConstructor() {
         r=1.0;
    }
     //Methods to return area
    public double area() {
        return 3.14 * r * r;
    }
}
```

Example(cont'd)

```
public class CircleConstructorTest{
    public static void main(String args[]){
        CircleConstructor circleA = new CircleConstructor(20.0);
        CircleConstructor circleB = new CircleConstructor(10.0);
        CircleConstructor circleC = new CircleConstructor();
        Double ca = circleA.area();
        System.out.println(ca);
        Double cb = circleB.area();
        System.out.println(cb);
        Double cc = circleC.area();
        System.out.println(cc);
    }
}
```

Accessors and Mutators

- A class's **private** fields can be manipulated only by methods of that class.
- 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 a value. *Get* methods are also commonly called accessor methods or **query methods**

Example

```
class Person{
   private String fname;
   private String mname;
   private String lname;
   private String addr;
   public Person(String  firstname, String  middlename, String  lastname,
    String  address){// constructor
      fname=firstname;
      mname=middlename;
      lname=lastname;
      addr=address;
   }
   public String getFname(){ // accessor for fname
     return fname;
   }
   public String getMname(){ // accessor for Mname
      return mname;
   }
```

Example(cont'd)

```
public String getLname(){ // acccessor for Lname
      return lname;
   }
   public String getAddress(){ // accessor for Address
      return addr;
   }
   public void setAddress(String address){ //mutator for address
      addr=address;
   }
   public void setLname(String lastname){ //mutator for Last name
      lname=lastname;
   }
}
```

Example(cont'd)

```
public class PersonExample{
   public static void main(String args[]){
      Person per=new Person("Alice","Bob", "James", "London");
      System .out.println(per.getFname()+" "+per.getMname()+"
    "+per.getLname()+ " "+per.getAddress()  );
      per.setAddress("Paris"); //to change London to Paris
      per.setLname("Baker"); //to change  last name from James to Baker
      System .out.println(per.getFname()+" "+per.getMname()+"
    "+per.getLname()+ " "+per.getAddress()  );
   }
}
```

The this Keyword

- The keyword **this** refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.

(I) Using this to Reference Hidden Data Fields

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Example

```
// Use this to resolve name-space collisions.
class Box {
   double width;
   double height;
   double depth;
   // This is the constructor for Box.
   Box(double width, double height, double depth) {
      this.width = width;
      this.height = height;
      this.depth = depth;
   }
   // compute and return volume
   double volume() {
      return width * height * depth;
   }
}
```

Example(cont'd)

```
class BoxDemo{
   public static void main(String args[]) {
      // declare, allocate, and initialize Box objects
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box(3, 6, 9);
      double vol;
      // get volume of first box
     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     // get volume of second box
     vol = mybox2.volume();
      System.out.println("Volume is " + vol);
   }
}
```

The this Keyword(cont'd)

- **(II) To call another overloaded Constructor**
  - The **this** keyword can be used to invoke another constructor of the same class
  - **Example**

```
class JBT {
   JBT() {
      this("JBT"); //call to this must be the first statement in constructor
      System.out.println("Inside Constructor without parameter");
   }
   JBT(String str) {
     System.out.println("Inside Constructor with String parameter as " + str);
   }
   public static void main(String[] args) {
        JBT obj = new JBT();
   }
}
```

Example: predict output

```
public class Officer{
    public Officer(){
        this("Second");
        System.out.println("I am First");
    }
    public Officer(String name){
        System.out.println("Officer name is " + name);
    }
    public Officer(int salary){
        this();
        System.out.println("Officer salary is " + salary);
    }
    public static void main(String args[]){
        Officer o1 = new Officer(9000);
    }
}
```

Activity

- Define a **class Product** with members(name, price, qty & calcCost(),calcTax())
  - Input values
    - Using constructor
    - Using input from keyboard(Scanner)
    - What if for N-products
- Define **four static methods** of simple calculator(Arithmetic)
  - Input values
    - Using constructor
    - Using input from keyboard(Scanner)

Packages

- **Packages** - a way of grouping similar types of classes /interfaces together.
- It is a great way to achieve **reusability.**
- We can simply **import** a class providing the required functionality from an existing package and use it in our program.
  - it avoids name conflicts and controls access of class, interface and enumeration etc.
  - It is easier to locate the related classes
  - consists of a lot of classes but only few needs to be exposed as most of them are required internally. Thus, we can **hide** the classes and prevent programs or other packages from accessing classes which are meant for internal usage only.

Cont'd…

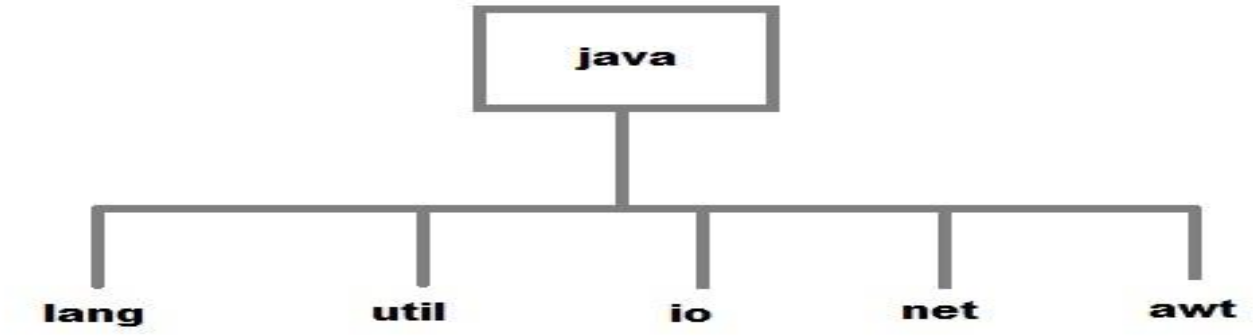
- The Packages are categorized as :
  - **Built-in packages**
    - These packages consists of a large number of classes which are a part of Java **API**
  - **Accessing classes in a package: E.g.:**
    - **import** java.util.Random; *// import the Random class from util package*
    - **import** java.util.*; *// import all the class from util package*
  - User-defined packages (packages defined by programmers to bundle group of related classes)
    - Java is a friendly language and permits to create our own packages and use in programming.
    - Packages avoid **name collision** problems.
    - Creating packages are indispensable in project development where number of developers are involved doing different modules and tasks.

Example: Built-in packages

Package Name	Description
java.lang	Contains classes that are fundamental to Java programming. This package is automatically imported.
java.io	Contains classes for supporting input / output operations.
java.util	Contains utility classes which implement data structures like Linked List, Hash Table, Dictionary, etc and support for Date / Time operations.
java.applet	Contains classes for creating Applets.
java.awt	Contains classes for implementing the components of graphical user interface (like buttons, menus, etc.).
java.net	Contains classes for supporting networking operations.

Subpackage

- A package created inside another package is known as a **subpackage**.
- When we import a package, subpackages are not imported by default. They have to be imported explicitly.
- E.g.:
  - import java.util.*; (util is a subpackage inside java package)
  - import javafx.stage.Stage;
  - import java.io.*;



User defined packages

- Creating a package in java is quite easy.
- Simply include a package command followed by name of the package as the first statement in java source file.
  - package RelationEg;
- However, because of new editors we can simply create using GUI wizard
  - Right click on your project
  - Point to New, then java-package
  - Give a name
  - Then you can create many-related classes in it

Example

```
package REL1;
public class Comp1 {
   public int getMax(int x, int y) {
      if ( x > y ) {
         return x;
      }
      else {
         return y;
      }
   }
}
```

Cont’d

```
package packageeg;
import REL1.Comp1;
public class EgComp {
   public static void main(String args[]) {
      int val1 = 7, val2 = 9;
      Comp1 comp = new Comp1();
      int max = comp.getMax(val1, val2); // get the max value
      System.out.println("Maximum value is " + max);
   }
}
```

Activity

- 1.Create user defined package calculator and define four methods in one class
  - Define a new class in different package and call the methods to do the four operations
    - Show simple import and static import
- 2. define **your own** packages and show userdefined package import
- 3. define a class Student {name, age, ht,wt,st-type}
  - Methods –insert(), display(),filter(), Undergraduate(),calcBMI()

Static Import

- It facilitates accessing any static member of an imported class directly i.e. without using the class name.

```
import static java.lang.Math.*;
public class Eg2 {
   public static void main(String args[]) {
      double val = 64.0;
      double sqroot = sqrt(val);
      System.out.println("Sq. root of " + val + " is " + sqroot);
   }
}
```

Exercise

- Define a class called CalAge. This class is used to calculate age of a person from her or his date of birth and the current date. Include a mutator method that allows the user to enter her or his date of birth and set the value for current date. Also include a method to return the age in years and months (for example, 25.5 years) as a double value. Include an additional method to check if the date of birth entered by the user is a valid one. For example, 30 February 2008 is an invalid date. Embed your class in a test program.
- Define a class called Journal that could be used to store an entry for a research paper that will be published. The class should have instance variables to store the author's name, title of the paper, and the date of submission. Add a constructor to the class that allows the user of the class to set all instance variables. Also add a method, displayDetails, that outputs all the instance variables, and another method called getSubmissionDetails that returns the title of the paper, with the first letter of each word capitalized. Test your class from the main method.

Read about…

- Wrapper Classes
- Creating JAR and Executable JAR Files