CS6308- Java Programming

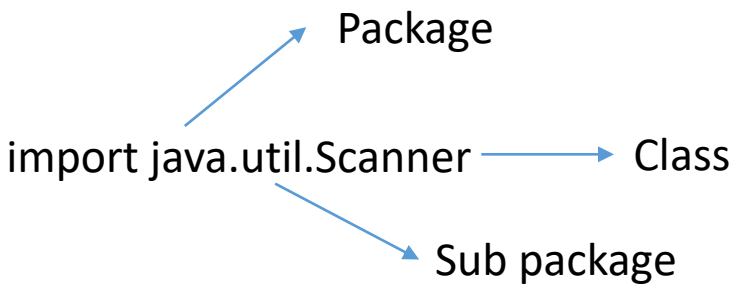
V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Syllabus

MODULE III JAVA OBJECTS – 2	L	T	P	EL
	3	0	4	3
Inheritance and Polymorphism – Super classes and sub classes, overriding, object class and its methods, casting, instance of, Array list, Abstract Classes, Interfaces, Packages, Exception Handling				
SUGGESTED ACTIVITIES : • flipped classroom • Practical - implementation of Java programs – use Inheritance, polymorphism, abstract classes and interfaces, creating user defined exceptions • EL – dynamic binding, need for inheritance, polymorphism, abstract classes and interfaces				
SUGGESTED EVALUATION METHODS: • Assignment problems • Quizzes				

Packages

- What is a package?
  - Package is a mechanism to group related classes and interfaces.
- Why packages?
  - Organize: To group related classes and interfaces
  - Reusability: To import the required class to use
  - Encapsulation: To control accessibility of the classes and interfaces
  - Name conflicts: To avoid name conflict for same class in different packages.



Types of packages

- User defined package:
  - The package user create is called user-defined package.
- Built-in package:
  - Predefined package like java.io.*, java.lang.*

Packages

- Packages are containers for classes.
- They are used to keep the class name space compartmentalized.
- For example, a package allows user to create a class named List, which can store in new package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- Without package:The name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions.
  - Need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
- The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package

Defining a Package

- Include a package command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The package statement defines a name space in which classes are stored.
  - Omit the package statement, the class names are put into the default package, which has no name.
- This is the general form of the package statement:
  - package pkg;
    - Here, pkg is the name of the package.
    - For example, the following statement creates a package called mypackage:
      - package mypackage;

Package

- The .class files for any classes being part of mypackage must be stored in a directory called mypackage.
- i.e directory name must match the package name exactly.
- More than one file can include the same package statement.
- Create a hierarchy of packages by use of a period.
  - The general form of a multileveled package statement is shown here:
  - package pkg1[.pkg2[.pkg3]];
  - a package declared as package a.b.c; needs to be stored in a\b\c in a Windows environment.

A Short Package Example

```
// A simple package
package mypack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }

  void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];
    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);

    for(int i=0; i<3; i++) current[i].show();
  }
}
```

Packages and Member Access

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- The class is Java's smallest unit of abstraction.
- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

Packages and Member Access

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Class Member Access

An Access Example

```
package p1;

public class Protection {
   int n = 1;
   private int n_pri = 2;
   protected int n_pro = 3;
   public int n_pub = 4;

   public Protection() {
      System.out.println("base constructor");
      System.out.println("n = " + n);
      System.out.println("n_pri = " + n_pri);
      System.out.println("n_pro = " + n_pro);
      System.out.println("n_pub = " + n_pub);
   }
}
```

This is file Protection.java:

```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

// class only
// System.out.println("n_pri = "4 + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file Derived.java:

```
package p1;

class SamePackage {
  SamePackage() {

    Protection p = new Protection();
    System.out.println("same package constructor");
    System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

This is file SamePackage.java:

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("derived other package constructor");

//  class or package only
//  System.out.println("n = " + n);

//  class only
//  System.out.println("n_pri = " + n_pri);

    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
```

This is file Protection2.java:

- The two classes defined in p2 cover the other two conditions that are affected by access control.
- The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri (because it is private) and n, the variable declared with the default protection.
- Default only allows access from within the class or the package, not extra-package subclasses.

```
package p2;

class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");

//  class or package only
//  System.out.println("n = " + p.n);

//  class only
//  System.out.println("n_pri = " + p.n_pri);

//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);

    System.out.println("n_pub = " + p.n_pub);
  }
}
```

This is file OtherPackage.java

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
  public static void main(String args[]) {
    Protection ob1 = new Protection();
    Derived ob2 = new Derived();
    SamePackage ob3 = new SamePackage();
  }
}
```

```
// Demo package p2.
package p2;

// Instantiate the various classes in p2.
public class Demo {
  public static void main(String args[]) {
    Protection2 ob1 = new Protection2();
    OtherPackage ob2 = new OtherPackage();
  }
}
```

Importing Packages

- All of the standard classes are stored in some named package.
- Classes within packages must be fully qualified with their package name or names.
- Java includes the import statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.
- This is the general form of the import statement:
  - import pkg1 [.pkg2].(classname | *);
  - pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.).
  - Finally, specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

```
import java.util.*;
class MyDate extends Date { }
```

```
class MyDate extends java.util.Date {
 }
```

```
package mypack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

```
import mypack.*;

class TestBalance {
  public static void main(String args[]) {

    /* Because Balance is public, you may use Balance
       class and call its constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);

    test.show(); // you may also call show()
  }
}
```

Summary

- A class can have only one package declaration but it can have more than one package import statements.

```
package package4; //This should be one
import package1;
import package2;
import package3;
```

- A class inside a package while importing another package then the package declaration should be the first statement, followed by package import.

```
package abcpackage;
import xyzpackage.*;
```

- Use the fully qualified name method when both the packages have a class with the same name,

```
package1. Example obj = new package1.Example();
package2. Example obj2 = new package2. Example();
```

```
//This will throw compilation error
import package1.*;
import package2.*;
```

Summary

- To import all the classes present in package and subpackage, we need to use two import statements like this:

```
import abc.*;
import abc.foo.*;
```

- import abc.*; will only import classes Example1, Example2 and Example3 of abc package but it will not import the classes of sub package.

```
import abc.*;
```

- To import the classes of subpackage you need to import like this:,

```
import abc.foo.*;
```

Abstract class

- A class that is declared using “**abstract**” keyword is known as abstract class.
- Abstract class can have abstract methods(methods without body) as well as concrete methods (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods.
- An abstract class can not be **instantiated**, which means you are not allowed to create an **object** of it.

Why an abstract class?

- A class `Animal` that has a method `sound()` and the subclasses of it like `Dog`, `Lion`, `Horse`, `Cat` etc.
- The animal sound differs from one animal to another, there is no point to implement this method in parent class.
- Every child class must override this method to give its own implementation details, like `Lion` class say “Roar” in this method and `Dog` class say “Woof”.
- All the animal child classes will and should override this method, then there is no point to implement this method in parent class.
- Thus, making this method abstract would be the good choice as by making this method abstract will force all the sub classes to implement this method(otherwise will get compilation error)
- The `Animal` class has an abstract method, must need to declare this class abstract.

```
//abstract parent class
 abstract class Animal{
      //abstract method
      public abstract void sound();
 }
//Dog class extends Animal class
public class Dog extends Animal{
     public void sound(){
       System.out.println("Woof");
      }
    public static void main(String args[]){
       Animal obj = new Dog();
      obj.sound();
    }
 }
```

Output:

```
Woof
```

Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration using abstract keyword
abstract class A{
   //This is abstract method
   abstract void myMethod();

   //This is concrete method with body
   void anotherMethod(){
      //Does something
   }
}
```

```
abstract class AbstractDemo{
   public void myMethod(){
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
        System.out.print("Abstract method");
   }
   public static void main(String args[])
   {
      //error: You can't create object of it
      AbstractDemo obj = new AbstractDemo();
      obj.anotherMethod();
   }
}
```

Output:

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo
```

Abstract class vs Concrete class

- An abstract class has no use until unless it is extended by some other class.
- An **abstract method** in a class must declare the class abstract as well. Concrete class don’t have abstract method . It’s vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
- Abstract class can have non-abstract method (concrete) as well.

Abstract class

- Abstract method

  1)Abstract method has no body.
  2) Always end the declaration with a **semicolon**(;).
  3) It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
  4) A class has to be declared abstract to have abstract methods.

```
abstract class MyClass{
   public void disp(){
     System.out.println("Concrete method of parent class");
   }
   abstract public void disp2();
}

class Demo extends MyClass{
   /* Must Override this method while extending
    * MyClas
    */
   public void disp2()
   {
       System.out.println("overriding abstract method");
   }
   public static void main(String args[]){
       Demo obj = new Demo();
       obj.disp2();
   }
}
```

Output:

```
overriding abstract method
```

Rule 1

- There are cases when it is difficult or often unnecessary to implement all the methods in parent class.
- In these cases, the parent class can be declared as abstract, which makes it a special class which is not complete on its own.
- A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Rule 2

- Abstract class cannot be instantiated which means you cannot create the object of it.
- To use this class, create another class that extends this class and provides the implementation of abstract methods, then can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

Rule 3

- If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.
- Abstraction is a process to show only “relevant” data and “hide” unnecessary details of an object from the user.
- Since abstract class allows concrete methods as well, it does not provide 100% abstraction. Abstract class provides partial abstraction.
- **Interfaces** on the other hand are used for 100% abstraction

Interfaces

- Using interface, can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default.

Why interface in java?

- To exhibit full abstraction.
  - Methods in interfaces do not have body, they have to be implemented by the class before accessing them.
  - The class that implements interface must implement all the methods of that interface.
- To support multiple inheritance
  - Java programming language does not allow to extend more than one class, However allow to implement more than one interfaces in the class.

Syntax

```
interface MyInterface
{
   /* All the methods are public abstract by default
    * As you see they have no body
    */
   public void method1();
   public void method2();
}
```

Example of an Interface in Java

```
interface MyInterface
{
    /* compiler will treat them as:
     * public abstract void method1();
     * public abstract void method2();
     */
    public void method1();
    public void method2();
}
class Demo implements MyInterface
{
    /* This class must have to implement both the abstract methods
     * else you will get compilation error
     */
    public void method1()
    {
         System.out.println("implementation of method1");
    }
    public void method2()
    {
         System.out.println("implementation of method2");
    }
    public static void main(String arg[])
    {
         MyInterface obj = new Demo();
         obj.method1();
    }
}
```

Output:

```
implementation of method1
```

This program, the class `Demo` only implements interface `Inf2`, however it has to provide the implementation of all the methods of interface `Inf1` as well, because interface Inf2 extends Inf1.

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
   public void method2();
}
public class Demo 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 Demo();
        obj.method2();
    }
}
```

Summary

- No instantiate for interface in java. That means object of an interface cannot be created.
- Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have a
- Abstract and concrete(methods with body) methods both.
- `implements` keyword is used by classes to implement an interface.
- While providing implementation in class of any method of an interface, it needs to be mentioned as public.
- Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.
- Interface cannot be declared as private, protected or transient.
- All the interface methods are by default **abstract and public**.

Summary

- Variables declared in interface are **public, static and final** by default.

```
interface Try {
 int a=10;
 public int a=10;
 public static final int a=10;
final int a=10;
static int a=0; }
```

- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try {
    int x;//Compile-time error }
```

Summary

- Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final.

```
class Sample implements Try {
 public static void main(String args[]) {
        x=20; //compile time error }
}
```

- An interface can extend any interface but cannot implement it.
- Class implements interface and interface extends interface.
- A **class** can implement any **number of interfaces**.

Summary

- If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
   public void aaa();
}
interface B
{
   public void aaa();
}
class Central implements A,B
{
   public void aaa()
   {
        //Any Code here
   }
   public static void main(String args[])
   {
        //Statements
    }
}
```

Summary

- A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
   public void aaa();
}
interface B
{
   public int aaa();
}

class Central implements A,B
{

   public void aaa() // error
   {
   }
   public int aaa() // error
   {
   }
   public static void main(String args[])
   {

   }
}
```

Summary

- Variable names conflicts can be resolved by interface name.

```
interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
       /* reference to x is ambiguous both variables are x
        * so we are using interface name to resolve the
        * variable
        */
       System.out.println(x);
       System.out.println(A.x);
       System.out.println(B.x);
    }
}
```