Core Java – Interfaces and Packages

Fresher Learning Program
January, 2012

People matter, results count.

Objectives of Interfaces and Packages

- Purpose:
  - Understanding of interfaces and packages and when and how to use them

- Product:
  - To understand what is interface and abstract class.
  - What package and why it is requried
  - Visibility criteria

- Process:
  - Theory Sessions followed by couple of assignments
  - A review at the end of the session and a Quiz.

Table of Contents

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Learning & Culture

Interfaces

Agenda

- What is an Abstract method and an Abstract class?
- What is Interface?
- Why Interface?
- Interface as a Type
- Interface vs. Class
- Defining an Interface
- Implementing an Interface
- Implementing multiple Interface's
- Inheritance among Interface's

What is an Abstract Class?

- **Abstract Methods**
  - ● Methods that do not have implementation (body)
  - ● To create an abstract method, just write the method declaration without the body and use the abstract keyword

For example,

```
// Note that there is no body
public abstract void someMethod();

public abstract void someMethod() { }
 // is not an abstract method; as there is body (which is { })
```

Abstract Class

- An abstract class is a class that contains one or more abstract methods

- An abstract class cannot instantiated

  // You will get a compile error on the following code

  MyAbstractClass a1 = new MyAbstractClass();

- Another class (Concrete class) has to provide implementation of the abstract methods

  – Concrete class has to implement all abstract methods of the abstract class in order to be used for instantiation

  – Concrete class uses extends keyword

Sample Abstract Class

```
public abstract class LivingThing {
        public void breath(){
                System.out.println("Living Thing breathing...");
        }

        public void eat(){
                System.out.println("Living Thing eating...");
        }

        /** Abstract method walk() */
        public abstract void walk();
}
```

Extending an Abstract Class

When a concrete class extends the *LivingThing* abstract class, it must implement the abstract method *walk(), or else, that subclass will also* become an abstract class, and therefore cannot be instantiated.

For example,

```
public class Human extends LivingThing {
        public void walk(){
                System.out.println("Human walks...");
        }
}
```

Interface - Basics

What is an Interface?

- It defines a standard and public way of specifying the behaviour of classes
- Defines a contract
- All methods of an interface are abstract methods – Defines the signatures of a set of methods, without the body (implementation of the methods)
- A concrete class must implement the interface (all the abstract methods of the Interface)
- It allows classes, regardless of their locations in the class hierarchy, to implement common behaviors

Interface - Definition

Example: Interface Relation

```
// Note that Interface contains just set of method
// signatures without any implementations.
// No need to say abstract modifier for each method
// since it assumed.


public interface Relation {
            public boolean isGreater( Object a, Object b);
            public boolean isLess( Object a, Object b);
            public boolean isEqual( Object a, Object b);
}
```

Interface vs. Abstract Class

- All methods of an Interface are abstract methods while some methods of an Abstract class are abstract methods
- Abstract methods of abstract class have abstract Modifier
- An interface can only define constants while abstract class can have fields
- Interfaces have no direct inherited relationship with any particular class, they are defined independently
- Interfaces themselves have inheritance relationship among themselves

Interface – implementation

- When a class uses an interface, it is done using the ‘implements’ keyword and that class is said to implement the specified interface.
- A class can implement more than one interface at a time.
- When a class implements an interface, it has to provide definitions to all the methods declared in the interface

Interface vs. Class: Commonality

- Interfaces and classes are both types – this means that an interface can be used in places where a class can be used
  - For example:

```
// Recommended practice
PersonInterface pi = new Person();

// Not recommended practice
Person pc = new Person();
```

- Interface and Class can both define methods

Defining Interfaces

To define an interface, we write:

```
public interface [InterfaceName] {
        //some methods without the body
}
```

Example:

```
public interface Relation {
        public boolean isGreater( Object a, Object b);
        public boolean isLess( Object a, Object b);
}
```

Interface – implementation

To create a concrete class that implements an interface, use the implements keyword.

```
// Line class implements Relation interface
public class Line implements Relation {
        private double x1;
        private double x2;
        private double y1;
        private double y2;
        public Line(double x1, double x2, double y1, double y2){
                this.x1 = x1;
                this.x2 = x2;
                this.y1 = y1;
                this.y2 = y2;
        }
}
```

Interface – variables

```
public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)* (y2-y1));
        return length;
}
public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen > bLen);
}
public boolean isLess( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();
        return (aLen < bLen);
}
```

Implementing Interfaces

When your class tries to implement an interface always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override abstract method
isGreater(java.lang.Object,java.lang.Object) in Relation
public class Line implements Relation
^
1 error
```

Relationship of an Interface to a Class

- A concrete class can only extend one super class, but it can implement multiple Interfaces
- The Java programming language does not permit multiple inheritance, but interfaces provide an alternative.
- All abstract methods of all interfaces have to be implemented by the concrete class
- A concrete class extends one super class but multiple Interfaces:

Example

```
public class ComputerScienceStudent extends Student implements
   PersonInterface,AnotherInterface,Third interface {
              // All abstract methods of all interfaces
              // need to be implemented.
}
```

Inheritance Among Interfaces

Interfaces are not part of the class hierarchy However, interfaces can have inheritance relationship among themselves

```
public interface PersonInterface {
        void doSomething();
}

public interface StudentInterface extends PersonInterface {
        void doExtraSomething();
}
```

Interface Uses

Why do we use Interfaces?

- To reveal an object's programming interface (functionality of the object) without revealing its implementation
- This is the concept of encapsulation
- The implementation can change without affecting the caller of the interface
- The caller does not need the implementation at the compile time
- It needs only the interface at the compile time
- During runtime, actual object instance is associated with the interface type

Interface Uses ……Contd…..

To have unrelated classes implement similar methods (behaviors)

– One class is not a sub-class of another

- Example:
  - – Class Line and Class MyInteger
- They are not related through inheritance
- You want both to implement comparison methods
  - – checkIsGreater(Object x, Object y)
  - – checkIsLess(Object x, Object y)
  - – checkIsEqual(Object x, Object y)
  - – Define Comparison interface which has the three abstract methods above

Interfaces & Abstract Class

- All methods of an Interface are abstract methods while some methods of an Abstract class are abstract methods

- Abstract methods of abstract class have abstract Modifier

- An interface can only define constants while abstract class can have fields.

- Interfaces have no direct inherited relationship with any particular class, they are defined independently.

- Interfaces themselves have inheritance relationship among themselves.

When to use an Abstract Class over Interface?

- For non-abstract methods, you want to use them when you want to provide common implementation code for all sub-classes.
  - – Reducing the duplication
- For abstract methods, the motivation is the same with the ones in the interface – to impose a common behavior for all sun-classes without dictating how to implement it.
- Remember a concrete can extend only one super class whether that super class is in the form of concrete class or abstract class.

Dynamic Method Dispatch

- Also known as Runtime Polymorphism
- Interfaces enables polymorphism, since program may call an interface method, and the proper version of that method will be executed depending on the type of object instance passed to the interface method call.
- A process in which a call to an overridden method is resolved at runtime rather than at compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Example

```
public interface myInterface {
    public void method();
}

public class SubClass1 implements myInterface {
    public void method(){
        System.out.println("Inside SubClass1");
    }
}

public class SubClass2 implements myInterface {
    public void method(){
        System.out.println("Inside SubClass2");
    }
}
```

Example

```
public class Test {

    public static void main(String args[]) {

        myInterface myInterface = new SubClass1();
        myInterface.method();

        myInterface = new SubClass2();
        myInterface.method();
    }
}
```

```
OUTPUT –
Inside SubClass1
Inside SubClass2
```

Packages

Packages - Basics

A package is a grouping of related types providing access protection and name space management

Note that types refers to classes, interfaces, enumerations, and annotation types.

Types are often referred to simply as classes and interfaces since enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to simply as classes and interfaces.

Benefits of Packaging

- You and other programmers can easily determine that these classes and interfaces are related.

- You and other programmers know where to find classes and interfaces that can provide graphics related functions.

- The names of your classes and interfaces won't conflict with the names in other packages because the package creates a new namespace.

- You can allow classes within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Placing a Class in a Package

To place a class in a package, we write the following as the first line of the code (except comments)

```
package <packageName>;
```

Example -

```
package myownpackage;
```

Using Classes from Other Packages

- To use a public package member (classes and interfaces) from outside its package, you must do one of the following

  – Import the package member using import statement

  – Import the member's entire package using import statement

  – Refer to the member by its fully qualified name (without using import statement)

Importing Packages

- To be able to use classes outside of the package you are currently working in, you need to import the package of those classes.
- By default, all your Java programs import the java.lang.* package, that is why you can use classes like String and Integers inside the program even though you haven‘t imported any packages.
- The syntax for importing packages is as follows:

  import <*nameOfPackage>;*

```
// Importing a class
import java.util.Date;

// Importing all classes in the java.util package
import java.util.*;
```

Access Modifier

Or

Visibility Criteria

Access Modifiers

- There are four different types of member access modifiers in Java:
  – public (Least restrictive)
  – protected
  – default
  – private (Most restrictive)

- The first three access modifiers are explicitly written in the code to indicate the access type, for the 3rd one (“default”), no keyword is used.

public accessibility

- public access – specifies that class members (variables or methods) are accessible to anyone, both inside and outside the class and outside of the package.

– Any object that interacts with the class can have access to the public members of the class.

– Keyword: public

```
public class StudentRecord {
        public int name;
        public String getName(){
                return name;
        }
}
```

Learning & Culture

default accessibility

- Default access – specifies that only classes in the same package can have access to the class' variables and methods
- No actual keyword for the default modifier; it is applied in the absence of an access modifier.

```
public class StudentRecord {
        // default access to instance variable
        int name;
        //default access to method
        String getName(){
                return name;
        }
}
```

private accessibility

- private accessibility – specifies that the class members are only accessible by the class they are defined in.

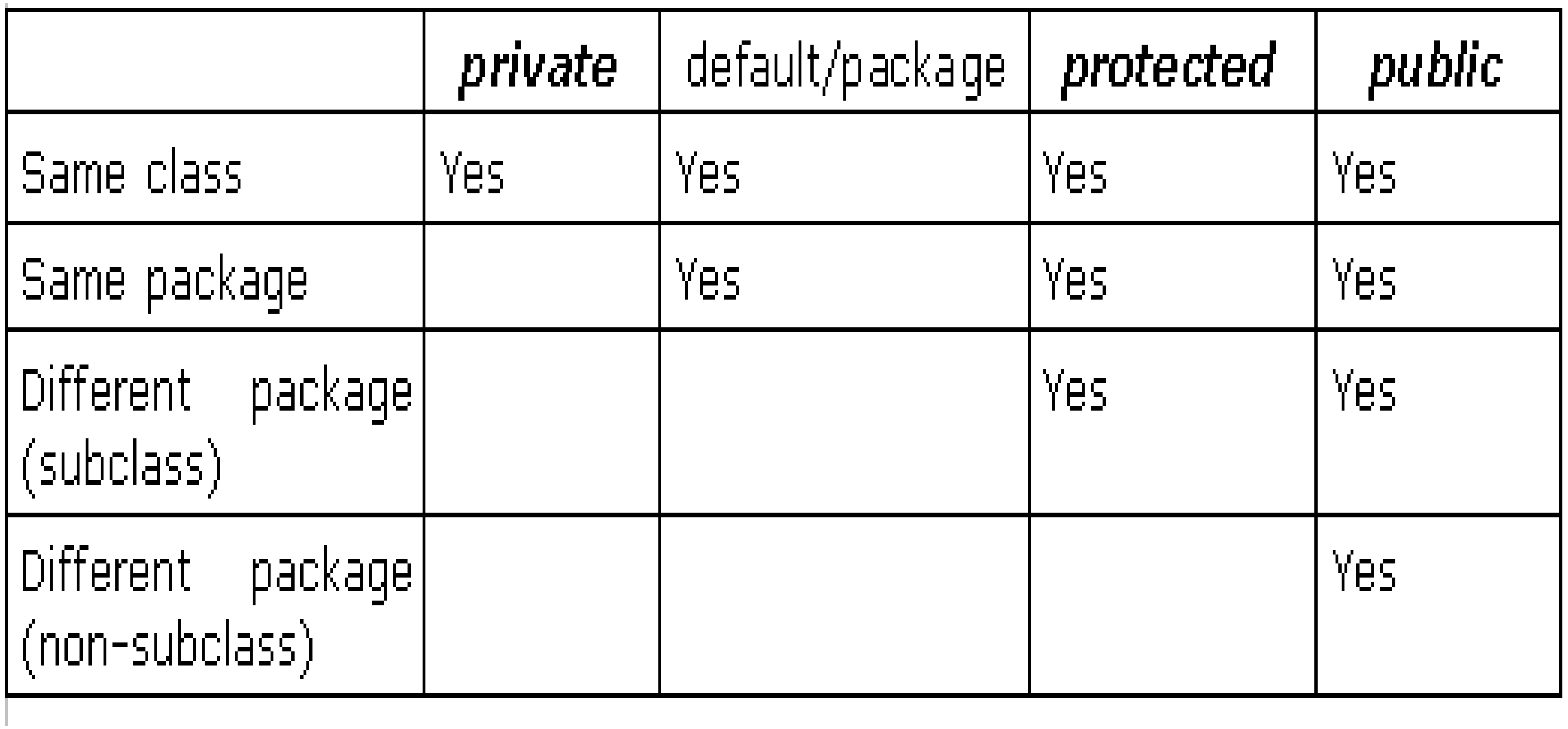
- Keyword: private

Ex:

```
public class StudentRecord {
        //default access to instance variable
        private int name;
        //default access to method
        private String getName(){
                return name;
        }
}
```

Java Program Structure:

- **The Access Modifiers**



	private	default/package	*protected*	*public*
Same class	Yes	Yes	Yes	Yes
Same package		Yes	Yes	Yes
Different package (subclass)			Yes	Yes
Different package (non-subclass)				Yes

Recap

Interface

access modifier

Package

Interface-variable

abstract class

abstract method

Thank You For Your Time

People matter, results count.