SLIIT
Discover Your Future

Object Oriented Programming

Week 04

Abstract Classes, Interfaces and Exception Handling

SLIIT
Discover Your Future

Learning Outcomes

At the end of the Lecture students should be able to apply the following concepts in the programs that you write.

- Abstract Classes
- Interfaces
- Exception Handling – Catching runtime Errors
- Packages
- Access Modifiers default

Abstract Classes

- Used in situations in which you will want to define a superclass where some methods don't have a complete implementation.
- Here we are expecting the sub classes will implement these methods.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. e.g an area() method of a Shape class.

Abstract Classes

```java
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();
  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}
class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}
```

AbstractClasses.java

Abstract Classes

- Refer the following class called Shape.

```
class Shape {
    public double getArea() {
        // How to Implement this code
    }
}
```

- Any Shape has an area. So that class shape can contain a method called getArea()

```
class Circle extends Shape {
  double radius;
  public double getArea() {
      return Math.PI * radius * radius;
  }
}
```

But how to calculate the area of a shape?
If we are asked to calculate the area of a circle, then we know how to calculate the area of a circle.

Abstract Classes

- getArea() method will have different implementations depending on the child class.
- Class Shape behave as a super class.
- Any shape has an area. So that class Shape can contain a method called getArea().

```
class Rectangle extends Shape {
  double width, height;
  public double getArea() {
      return width * height;
  }
}
```

- But implementing getArea() method is possible only when we know the child class.
- The implementation of getArea() method is different from one child class to another.

Abstract Class

- The methods which cannot be implemented MUST be defined as abstract.

```
abstract class Shape {
    abstract public double getArea();
}
```

- An abstract method does not have a method implementation (not even { }).
- We cannot invoke (call) an abstract method.
- The class which contain at least one abstract method MUST be defined
- Every child class MUST override all the abstract methods of the parent class.
- If a child class did not override at least one abstract method of the parent, then the child class also become abstract.

SLIIT
Discover Your Future

Abstract Classes

- A class which contain at least one abstract method must be defined as abstract.
- Cannot create instances (object) from an abstract class.
- All the child classes must override abstract methods of the parent.
- Force the child class to contain methods defined by the parent class.
- Their purpose is to behave as parent (base) classes.
- Abstract classes are very generic
- Usually a class hierarchy is headed by an abstract class
- Classes from which objects can be instantiated - concrete
- The ability to create abstract methods is powerful – each new class that inherits is forced to override these methods. ex. Mouse clicks and drags

Abstract Classes

- An abstract class can behave as a data type.

```
Shape s1;
```

Correct

- But cannot create instances (object) from an abstract class.

```
s1 = new Shape();
```

Wrong

- But super class variables can refer to child class objects.

```
s1 = new Circle();
Shape s2 = new Rectangle()
```

Correct

- Can call the getArea() method belongs to the child.

```
s1.getArea();
s2.getArea();
```

Correct
But the two method calls will perform two different actions

Another Example

- Suppose we wanted to create a class GeometricObject
  - Reasonable concrete methods include
    - `getPosition()`
    - `setPosition()`
    - `getColor()`
    - `setColor()`
    - `paint()`
  - We can implement all the methods, but not `paint()`.
  - For `paint()`, we must know what kind of object is to be painted. Is it a square, a triangle, etc.
  - Method `paint()` should be an abstract method

Another Example

Example

Makes GeometricObject an abstract class

```
import java.awt.*;

abstract public class GeometricObject {
   // instance variables
   Point position;
   Color color;

   // getPosition(): return object position
   public Point getPosition() {
       return position;
   }
   // setPosition(): update object position
   public void setPosition(Point p) {
       position = p;
   }
```

Another Example

Example (continued)

Indicates that an implementation of method paint() will not be supplied

```
    // getColor(): return object color
    public Color getColor() {
        return color;
    }
    // setColor(): update object color
    public void setColor(Color c) {
        color = c;
    }
    // paint(): render the shape to graphics context g
    abstract public void paint(Graphics g);
}
```

Interfaces

- Is a contract between a class and the outside world.
- When a class implements an interface it promises to provide the behavior in the interface (Implement the methods specified in the interface).
- Interfaces can also store constants. These are essentially public static final variables.

Interfaces

```java
interface Callback {
  void callback(int param);
}

class Client implements Callback {
        // Implement Callback's interface
   public void callback(int p) {
     System.out.println("callback called with " + p);
   }
   void nonIfaceMeth() {
     System.out.println("Classes that implement interfaces " +
                        "may also define other members, too.");
    }
}
```

Interface.java

Interfaces

- Are fully abstract classes.
- Similar to classes, but
  All the methods in an interface are defined with out a body.
- (automatically abstract)
- Cannot contain class variables. But can contain constants (final variables)

```
interface Inter1 {
    int COUNT = 10;
    void test1();
    void test2();
}
```

Interfaces

```
class Test implements Inter1 {
    public void test1() {
    }
    public void test2() {
    }
}
```

Must Implement all methods in Interface (otherwise should be defined as an abstract class)

Access level must be public

Interfaces

- A Class can implement any number of interfaces.

```
class Test implements Inter1, Inter2, Inter3 {
```

- Then the class MUST override all the methods mentioned in all the interfaces.
- If the class did not override at least one method of the interfaces then the class become abstract.

Interfaces

- Java do not allow multiple inheritance.

✘
```
class PartTimeStudent extends Employee, Student {
```

- Classes can extend exactly one class and implement any number of interfaces.

✓
```
class PartTimeStudent implements Employee, Student {
```

- Using interfaces we can achieve some thing similar to multiple inheritance.
- An interface can extends another interface.

Interfaces

- Interfaces are designed to support dynamic method resolution at run time.
- it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

SLIIT
Discover Your Future

Abstract classes vs Interfaces

- Use an abstract class if
  - You want to share the code.
  - Expect to have common methods or properties
  - You want to have access modifiers other than public
  - You want to have properties which not static or not final
- Use an interface if
  - You want Unrelated classes to implement the interfaces.
  - Want to take advantages of multiple inheritances

SLIIT
Discover Your Future

Data access control

- **Member access modifiers** – control access to class members through the keywords **public, protected , default (friendly, no specifier) and private**
- **public –** members declared as public are accessible from anywhere in the program when the object is referenced.
- **protected –** subclasses can access protected method/data from the parent class.
- **Default –** such a class, method, or field can be accessed by a class inside the same package only.
- **private –** members declared as private are accessible *ONLY* to methods of the class in which they are defined. All private data are always accessible through the methods of their own class.
- **USUALLY: data (instance variables)** are declared **private, methods** are declared **public.**
- **NOTE:** using **public data** is uncommon and **dangerous** practice

SLIIT Discover Your Future

Controlling access – Class member

Member Restriction	this	Subclass	Package	General
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	—
`default`	✓	—	✓	—
`private`	✓	—	—	—

Here a member is either a property or a method.

Controlling access – Full Class

Member Restriction	this	Subclass	Package	General
`public`	✓	✓	✓	✓
`default`	✓	—	✓	—

There is no meaning for private or protected classes

Java Packages

- *Packages* are containers for classes. They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named **List**, which you can store in your own 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.

Java Packages

- The Java Package name consists of words separated by periods. The first part of the name represents the organization which created the package. The remaining words of the Java Package name reflect the contents of the package. The Java Package name also reflects its directory structure.
- Importing a Class from an Existing Java Package Through the import statement.
  * - all classes
- Example **`import java.io.*;`**

  `import java.awt.Font`;
- CANNOT import a package, only a class:

  `import java.io;` - **wrong!**

Default Packages

- If a class is in the same package as the class that uses it, it does not need **import** statement.
- **Default package -** all the complied classes in the current directory. If a package is not specified, the class is placed in the default package.

SLIIT
Discover Your Future

Java Standard Packages

Package Name	Description
java.lang	Contains language support classes (for e.g classes which defines primitive data types, math operations, etc.) . 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.

Creating your own packages

- Lets see how a class called Student and StudentApp can be created in the following packages.

```
package MyPack;
```

- Compiled classes should be stored in MyPack folder

```
C:\myjavaprgs\MyPack
C:\myjavaprgs>java MyPack.StudentApp
```

```
package sliit.foc.se;
```

- Compiled classes should be stored in followng folder

```
C:\myjavaprgs\sliit\foc\se
C:\myjavaprgs>java
sliit.foc.se.StudentApp
```

If the client program is not in the same package, you have to use the import command or use the class name with the package name.