Module 4

Packages and Interfaces, Exception Handling

Packages and Interfaces:

- Packages,
- Access Protection,
- Importing Packages,
- Interfaces,

Exception Handling:

- Exception-Handling Fundamentals,
- Exception Types,
- Uncaught Exceptions,
- Using try and catch,
- Multiple catch Clauses,
- Nested try Statements,
- throw, throws, finally,
- Java's Built-in Exceptions,
- Creating Your Own Exception Subclasses,
- Chained Exceptions,
- Using Exceptions.

Packages and Interfaces:

Packages:

- 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.
- **Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.**
- 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. You can also define class members that are only exposed to other members of the same package.

Defining a Package:

- To create a package is quite easy: simply 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. If you 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;

- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong.

- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as

package java.awt.image;

Finding Packages and CLASSPATH:

- Packages are mirrored by directories.
- The Java run-time system know where to look for packages that you create
  - ➔ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - ➔ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
  - ➔ Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

- For example, consider the following package specification:

package MyPack

can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

- When the second two options are used, the class path *must not* include **MyPack**, itself.
- It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is

C:\MyPrograms\Java\MyPack

- Then the class path to **MyPack** is

C:\MyPrograms\Java

A Short Package Example:

```
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[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();
        }
}
```

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
- Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory.
- Then, try executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

Access Protection:

- Packages add another dimension to access control.

- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages 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.
- Because of the interplay between classes and packages, 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
- The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.
- Table sums up the interactions.

	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

Importing Packages:

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package.

- 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. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

This is the general form of the **import** statement:

import *pkg1*[.*pkg2*].(*classname*|*);

Here, *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 (**.**).

- There is no practical limit on the depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.
- This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

- If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes.
- In that case, you will get a compile-time error and have to explicitly name the class specifying its package.
- It must be emphasized that the **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy.

- For example, this fragment uses an import statement:

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

The same example without the **import** statement looks like this:

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

In this version, **Date** is fully-qualified.

Interfaces:

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- That is, using **interface**, you 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 their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.

Defining an Interface:

- An interface is defined much like a class.
- This is the general form of an interface:

***access* interface *name* {**

***return-type method-name1*(*parameter-list*);**

***return-type method-name2*(*parameter-list*);**

type final-varname1 = value;

type final-varname2 = value;

// ...

***return-type method-nameN*(*parameter-list*);**

type final-varnameN = value;

}

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other code.
- In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- *name* is the name of the interface, and can be any valid identifier.
- Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list.
- They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations.
- They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public**.

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter.

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

Implementing Interfaces:

- Once an **interface** has been defined, one or more classes can implement that interface.

- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [*,interface...*]] {

// class-body

}

- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }
}
```

- Notice that **callback( )** is declared using the **public** access specifier.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
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.");
    }
}
```

Accessing Implementations Through Interface References:

- You can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the "callee."
- The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
        public static void main(String args[]) {
                Callback c = new Client();
                c.callback(42);
        }
}
```

The output of this program is shown here:

```
callback called with 42
```

- While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference.
- To sample this usage, first create the second implementation of **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
```

```
        }
}
```

Now, try the following class:

```
class TestIface2 {
        public static void main(String args[]) {
                Callback c = new Client();
                AnotherClient ob = new AnotherClient();
                c.callback(42);
                c = ob; // c now refers to AnotherClient object
                c.callback(42);
        }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Partial Implementations:

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- For example:

```
abstract class Incomplete implements Callback {
        int a, b;
        void show() {
                System.out.println(a + " " + b);
        }
        // ...
}
```

- Here, the class **Incomplete** does not implement **callback( )** and must be declared as abstract.
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Nested Interfaces:

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**.
- This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
class A {
        // this is a nested interface
        public interface NestedIF {
                boolean isNotNegative(int x);
        }
}
// B implements the nested interface.
class B implements A.NestedIF {
        public boolean isNotNegative(int x) {
                return x < 0 ? false : true;
        }
}
class NestedIFDemo {
        public static void main(String args[]) {
        // use a nested interface reference
                A.NestedIF nif = new B();
                if(nif.isNotNegative(10))
                System.out.println("10 is not negative");
                if(nif.isNotNegative(-12))
                System.out.println("this won't be displayed");
        }
```

}

- Notice that **A** defines a member interface called **NestedIF** and that it is declared **public**.
- Next, **B** implements the nested interface by specifying implements A.NestedIF
- Notice that the name is fully qualified by the enclosing class' name.
- Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**, this is legal.

Applying Interfaces:

- To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack.
- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**.

This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
        void push(int item); // store an item
        int pop(); // retrieve an item
}
class FixedStack implements IntStack {
        private int stck[];
        private int tos;
        // allocate and initialize stack
        FixedStack(int size) {
                stck = new int[size];
                tos = -1;
        }
        // Push an item onto the stack
        public void push(int item) {
                if(tos==stck.length-1) // use length member
                        System.out.println("Stack is full.");
```

```
                else
                        stck[++tos] = item;
        }
        // Pop an item from the stack
        public int pop() {
                if(tos < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return stck[tos--];
        }
}
class IFTest {
        public static void main(String args[]) {
                FixedStack mystack1 = new FixedStack(5);
                FixedStack mystack2 = new FixedStack(8);
                // push some numbers onto the stack
                for(int i=0; i<5; i++) mystack1.push(i);
                for(int i=0; i<8; i++) mystack2.push(i);
                // pop those numbers off the stack
                System.out.println("Stack in mystack1:");
                for(int i=0; i<5; i++)
                System.out.println(mystack1.pop());
                System.out.println("Stack in mystack2:");
                for(int i=0; i<8; i++)
                System.out.println(mystack2.pop());
        }
}
```

Variables in Interfaces:

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- It is as if that class were importing the constant fields into the class name space as **final** variables.
- The next example uses this technique to implement an automated "decision maker":

```
import java.util.Random;
interface SharedConstants {
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
class Question implements SharedConstants {
        Random rand = new Random();
        int ask() {
                int prob = (int) (100 * rand.nextDouble());
                if (prob < 30)
                return NO; // 30%
                else if (prob < 60)
                return YES; // 30%
                else if (prob < 75)
                return LATER; // 15%
                else if (prob < 98)
                return SOON; // 13%
                else
                return NEVER; // 2%
        }
}
class AskMe implements SharedConstants {
        static void answer(int result) {
```

```
                switch(result) {
                case NO:
                System.out.println("No");
                break;
                case YES:
                System.out.println("Yes");
                break;
                case MAYBE:
                System.out.println("Maybe");
                break;
                case LATER:
                System.out.println("Later");
                break;
                case SOON:
                System.out.println("Soon");
                break;
                case NEVER:
                System.out.println("Never");
                break;
        }
}
public static void main(String args[]) {
                Question q = new Question();
                answer(q.ask());
                answer(q.ask());
                answer(q.ask());
                answer(q.ask());
        }
}
```

Interfaces Can Be Extended:

- One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes.

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
        void meth1();
        void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
        void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
        public void meth1() {
                System.out.println("Implement meth1().");
        }
        public void meth2() {
                System.out.println("Implement meth2().");
        }
        public void meth3() {
                System.out.println("Implement meth3().");
        }
}
class IFExtend {
        public static void main(String arg[]) {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```