Session 3:

Outline

- Interfaces
- Packages
- Access Control Rules

Interfaces

Interface

- A Java *interface* is a collection of constants and abstract methods
  - since all methods in an interface are abstract, the abstract modifier is usually left off
- It has no constructor – no instances of an interface
- It can be implemented by **more than one class** in **different parts of the class hierarchy**
- Use the implements keyword

Defining An Interface

- The word **interface** is used instead of **class.**
- All methods are implicitly **public** and **abstract.**
- All variables are implicitly **final**, **public** and **static** i.e. constants
- The general form of the interface definition is:

```
interface Name {

        // constant definitions
        // method definitions
}
```

Interface: Declaration

interface is a reserved word

```
public interface Doable
{
        public static final String NAME;
        public static final double PI = 3.142;
        public static final int MAXIMUM = 300;

        public void doThis();
        public int doThat();
}
```

No method in an interface has a definition (body)

A semicolon immediately follows each method header

Implementing Interfaces

- Classes that implement interfaces must **define all interface methods with exact signature, or be abstract or the compiler will produce errors.**
- Classes must be **public** or with **no access modifier.**
- Methods that implement interface methods must be **public.**
- Method signatures of implementing class must be **exactly the same as interface.**
- A class can **implement more than one interface**, BUT it can **extend** (inherit from) **only one** class

Implementing Interfaces (cont'd)

```
public class Something implements Doable
{
        public void doThis ()
        {
          // whatever
        }

        public void doThat ()
        {
          // whatever
        }

  // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Extending Interfaces

- An interface may also **extend** other interfaces.

Example:

```
interface  Book1{
        int  print1();
    }

    interface Book2 extends Book1{
        int print2();
    }
```

- Anything that contracts to implement Book2 *must* implement both print1 and print2

Interface Hierarchies

- **Inheritance** can be applied to interfaces as well as classes
  - *superinterface* and *subinterface*

- **Multiple inheritance allowed**
  - ☑ An interface may extend multiple interfaces.
  - ☑ A class my implement multiple interfaces : the interfaces are listed in the implements clause, separated by commas.

Interface Example

```
public interface Employee
{
    public final String company = "ACME";
    public void hire( int salary );
    public void promote();
}
```

cont'd

```
public class RegularEmployee implements Employee{
        String firstName;
        String lastName;
        int salary;
      int positionLevel;
      public RegularEmployee( String fn, String ln ){
          firstName = fn;
          lastName = ln;
          salary = 0;
          positionLevel = 0;
      }
```

cont'd

```
    public void hire( int sal ){
        salary = sal;
    }
    public void promote(){
        positionLevel++;
        salary += 1000;
    }
    public void printInfo(){
        System.out.print("Employee: "+firstName+" "+lastName+" ");
        System.out.println("Salary: "+salary+" Level: "+positionLevel);
    }
}
```

cont'd

```
public class Manager implements Employee{
    String firstName;
    String lastName;
    int salary;
    int positionLevel;
    public Manager( String fn, String ln ){
        firstName = fn;
        lastName = ln;
        salary = 0;
        positionLevel = 10;
    }
```

cont'd

```
    public void hire( int sal ){
        salary = sal;
    }
    public void promote(){
        positionLevel++;
        salary += 5000;
    }
    public void printInfo(){
        System.out.print("Manager: "+firstName+" "+lastName+" ");
        System.out.println("Salary: "+salary+" Level:"+positionLevel);
    }
}
```

cont'd

```
public class UseEmployee
{   public static void main( String args[] )
    {
        Manager m = new Manager("Scott", "McNealy");
        m.printInfo();
        m.hire( 10000 );
        m.promote();
        m.printInfo();
        RegularEmployee re = new RegularEmployee("James“,"Gosling");
        re.printInfo();
        re.hire( 10000 );
        re.promote();
        re.printInfo();
```

cont'd

```
        Employee e = m;
        e.promote();
        ((Manager)e).printInfo();
        e = re;
        re.promote();
        ((RegularEmployee)e).printInfo();
    }
}
```

Interfaces: Java Standard Class Library

- The Java Standard Class library defines many interfaces:
  - the ***Iterator*** interface contains methods that allow the user to move through a collection of objects easily

    hasNext(), next(), remove()

  - the ***Comparable*** interface contains an abstract method called **compareTo**, which is used to compare two objects

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

Online Java Class Libraries

Interfaces

Classes

java.util (Java 2 Platform SE 5.0) - Windows Internet Explorer

http://java.sun.com/j2se/1.5.0/docs/api/

File Edit View Favorites Tools Help

java.util (Java 2 Platform SE 5.0)

java.text
java.util
java.util.concurrent
java.util.concurrent.atomic
java.util.concurrent.locks
java.util.jar
java.util.logging
java.util.prefs
java.util.regex
java.util.zip
javax.accessibility
javax.activity

java.util

Interfaces
Collection
Comparator
Enumeration
EventListener
Formattable
Iterator
List
ListIterator
Map
Map.Entry
Observer
Queue
RandomAccess
Set
SortedMap
SortedSet

Classes
AbstractCollection
AbstractList
AbstractMap
AbstractQueue
AbstractSequentialList
AbstractSet
ArrayList

Overview Package Class Use Tree Deprecated Index Help

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Java™ 2 Platform Standard Ed. 5.0

Package java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

See:
Description

Interface Summary	
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The `Formattable` interface must be implemented by any class that needs to perform custom formatting using the `'s'` conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
Observer	A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.
Queue<E>	A collection designed for holding elements prior to processing.
RandomAccess	Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access.
Set<E>	A collection that contains no duplicate elements.

http://java.sun.com/j2se/1.5.0/docs/api/index.html

Polymorphism via Interfaces

- Define a polymorphism reference through interface
  - declare a reference variable of an interface type

```
Doable obj;
```

  - the obj reference can be used to point to any object of any class that implements the Doable interface
  - the version of doThis depends on the type of object that obj is referring to:

```
obj.doThis();
```

More Examples

```
Speaker guest;

guest = new Philosopher();
guest.speak();

guest = Dog();
guest.speak();
```

```
Speaker special;
special = new Philosopher();

special.pontificate();// compiler error
```

```
Speaker special;
special = new Philosopher();

((Philosopher)special).pontificate();
```

```
public interface Speaker
{
   public void speak();
}

class Philosopher extends Human
   implements Speaker
{
   //
   public void speak()
   {…}
   public void pontificate()
   {…}
}

class Dog extends Animal
   implements Speaker
{
   //
   public void speak()
   {
   …
   }
}
```

Cast Object References

```
class Student { ... }
        class Undergraduate extends Student { ... }
                class Graduate extends Student { ... }

Student student1, student2;
student1 = new Undergraduate(); // ok
student2 = new Graduate();      // ok

Graduate student3;
student3 = student2; // compilation error

student3 = (Graduate) student2; // explicit cast, ok

student3 = (Graduate) student1;      // compilation ok, run-time error
```

Packages

Packages

- A ***package*** is a container or a collection of classes providing *access protection* .
- Code for all of a package's classes lives in a single directory.
- Dots in a package name correspond to subdirectories.
  - E.g. java.aw.image = java/awt/image
  - There is no connection between a package and its subpackages.

Why do we need Packages?

- Main reason to use package is to guarantee the uniqueness of class names.
- One can easily determine that these types are related.
- One knows where to find types that can provide task-related functions.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- One can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating Your Own Packages

- Each package class must be stored in a file in an appropriately named directory.
- The source code file for each package class must contain a package statement as its first non-commented statement.
- Only one package declaration per source file.
- Several packages can be stored in the same directory.
- Classes in different directories cannot be part of the same package.
- If no package is declared, then the class “belongs” to the default package.

Creating Packages

- To create a package:
  1. Create a subdirectory with the same name as the desired package and place the source files in that directory.
  2. Add a package statement to each file

     package packagename;
  3. Files in the main directory that want to use the package should include

```
import packagename.classname*; // to allow unqualified references to
                               // a particular package class
```

or

```
import packagename.*;     //Include all the classes in package
```

Import Examples

- This code

```
java.util.Date d =
  new java.util.Date();

java.awt.Point p =
  new java.awt.Point(1,2);

java.awt.Button b =
  new java.awt.Button();
```

- Can be abbreviated

```
import java.util.date;
Import java.awt.*;
…
Date d = new Date();
Point p = new Point(1,2);
Button b = new Button();
```

Package Example

```
//in the Shape.java file
public abstract class Shape {
    . . .
}

//in the Circle.java file
public class Circle extends Shape {
    . . .
}

//in the Rectangle.java file
public class Rectangle extends Shape {
    . . .
}
```

Package graphics: 1st step

- Choose a name for the package (**graphics**, for example) and put a package statement with that name at the top of every source file that contains the classes that you want to include in the package.

In the Shape.java file:

```
package graphics;
public abstract class Shape {
    . . .
}
```

…

In the Rectangle.java file:

```
package graphics;
public class Rectangle extends Shape
{
    . . .
}
```

Package graphics: 2nd step

- Put the source files in a directory whose name (graphics, for example) reflects the name of the package to which the type belongs:

...\graphics\Shape.java

...\graphics\Circle.java

...\graphics\Rectangle.java

...\graphics\Cylinder.java

etc.

How to use packages

1. Referring to a package member by its qualified name:

```
graphics.Circle myCircle = new graphics.Circle();
```

2. Importing a package member:

```
import graphics.Circle;
...
Circle myCircle = new Circle();
graphics.Rectangle myR = new graphics.Rectangle();
```

3. Importing an entire package:

```
import graphics.*;
...
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();
```

Some Predefined Java Packages

Package Name	Contents
java.applet	Classes for implementing applets
java.awt	Classes for graphics, windows, and GUI's
java.awt.event	Classes supporting AWT event handling
java.awt.image	Classes for image handling
java.awt.peer	Interface definitions s for platform independent graphical user interfaces (GUI's)
java.io	Classes for input and output
java.lang	Basic language classes like Math (always available in any Java program)
java.net	Classes for networking
java.util	Useful auxiliary classes like Date

Package and Protected Access

- **package access** means that a name is available everywhere in the same package (the same directory).

- **protected access** means that a name is available everywhere in the same package (the same directory), but also to any subclasses, wherever they may be protected access is “more public” than package access (default).

The protected Modifier

pacakge p1

class C1

protected int x

class C3

C1 c1;
c1.x can be read or modified

pacakge p2

class C2 extends C1

x can be read or modified in C2

class C4

C1 c1;
c1.x cannot be read nor modified

Package Example

```
package pack1;
public class PkgClass {
    public int publicInt;
    private int privateInt;
    int packageInt; // Package access
    public PkgClass(int pu, int pr, int pa) {
                publicInt= pu;
                privateInt= pr;
                packageInt= pa;
    }
    public void publicPrint() {
            System.out.println("Public");
    }
    private void privatePrint() {
            System.out.println("Private");
    }
    void packagePrint() { // Package access
            System.out.println("Package");
    }
}
```

Package Example

```
package pack1; // In same package
public class PkgTest {
    public static void main(String[] args) {
        PkgClass object1= new PkgClass(1, 2, 3);
        int pu= object1.publicInt;
✘       int pr= object1.privateInt;      No Access
        int pa= object1.packageInt;

        object1.publicPrint();
✘       object1.privatePrint();          No Access
        object1.packagePrint();
        System.exit(0);
    }
}       // Which statements will not compile?
```

cont'd

```
package pack2; // Different package (or none)
import pack1.*; // Import desired package

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

        PkgClass object1= new PkgClass(1, 2, 3);
        int pu= object1.publicInt;
        int pr= object1.privateInt;          No Access
        int pa= object1.packageInt;          No Access
        object1.publicPrint();
        object1.privatePrint();              No Access
        object1.packagePrint();              No Access
        System.exit(0);
    }
}

// Which statements will not compile?
```

Access Control Rules

Modifiers Overview

- **Modifiers** are Java keywords that give the compiler information about the nature of code, data, or classes.
- Modifiers may be applied to a class, a method or a variable.
- In Java, we accomplish encapsulation through the use of visibility modifiers.
- All the modifiers are:

Access	public, protected, private
Others	final , abstract, static, native, transient, synchronized, volatile

Access Modifiers Overviews

- Access modifiers control which classes may use the following feature: the class itself, its class variables, its methods and constructors, its nested class.
- Access modifiers are: ***public, protected and private*.**
- The *local variable* ***must not have access*** modifier.
- You can have only ONE access modifier to apply to a feature.
- The *class can only be defied as* ***public or*** **<default>.**
- If NO access modifier is used, then ***default*** access level is assigned.

Access Modifiers - *Public*

- The most generous access modifier, a public class, variable, or method may be used in any Java program without restriction.

- The only access modifier permitted to top-level classes is public, there is no such thing as a protected or private top-level class.
  example:

```
public class Book {…}
```

Default access level can be applied to top-level classes
example:

```
class Book {…}
```

Access modifier *public*

package p1

class C1

public int x

class C3

C1 c1;
c1.x **can** be read or modified

package p2

class C2 extends C1

x **can** be read or modified in C2

class C4

C1 c1;
c1.x **can** be read nor modified

Access Modifier - *private*

- The least generous access modifier , a private variable or method may only be used by an instance (the real one) of the class that declares the variable or method.

- In an ideal program, most or all of the variables of a class are kept private.

Access modifier *private*

package p1

class C1

private int x

class C3

C1 c1;
c1.x **cannot** be read or modified

package p2

class C2 extends C1

x **cannot** be read or modified in C2

class C4

C1 c1;
c1.x **cannot** be read nor modified

Access Level - Default

- Also known as package access. Default access is applied if you **don't** specify an access modifier. It can be applied to data, methods and classes.

- Java runtime environment considers that all class files in its current working directory constitute a package.

- A class' default features are accessible to any class in the same package as the class. Classes outside the package may **not** access the default features, because the features are default, not public.

- Classes outside the package may subclass the classes in the package; however, even the subclasses may not access the default features, because the features are default. (Note that default is a keyword in Java, using in switch statement.)

Access modifier *Default*

package p1

class C1

int x

class C3

C1 c1;
c1.x **can** be read or modified

package p2

class C2 extends C1

x **cannot** be read or modified in C2

class C4

C1 c1;
c1.x **cannot** be read nor modified

Access Modifier - protected

- A protected feature of a class is available to all classes in the same package, just like a default feature.
- Moreover, a protected feature of a class is available to all subclasses of the class that owns the protected feature.
- This access is provided even to subclasses that reside in a different package.
- protected access modifier can be applied to variables, methods, and inner classes.

Access modifier *protected*

package p1

class C1

protected int x

class C3

C1 c1;
c1.x **can** be read or modified

package p2

class C2 extends C1

x **can** be read or modified in C2

class C4

C1 c1;
c1.x **cannot** be read nor modified

Recommended Access Levels

- Instance and static fields: Always **private**.
- Methods: **public** or **private**
- Classes and interfaces: **public** or **default**
- In general, inner classes should **not** be **public**

Access Modifier Level

Data Fields and Methods	Modifiers			
	public	protected	default	private
Accessible from same class?	yes	yes	yes	yes
Accessible to classes (nonsubclass) from the **same package?**	yes	yes	yes	no
Accessible to classes (**nonsubclass)** from **different package?**	yes	no	no	no
Accessible to subclasses from **different package?**	yes	no	no	no
Inherited by subclass in the same package?	yes	yes	no	no
Inherited by subclass in different package?	yes	yes	no	no

Thank you!