Interfaces and Packages

What is an Interface?

- An *interface* defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.
- An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.
- An *interface* is a named collection of method declarations (without implementations).
- Interface reserve behaviors for classes that implement them.

- Methods declared in an interface are always public and abstract, therefore Java compiler will not complain if you omit both keywords
- Static methods cannot be declared in the interfaces – these methods are never abstract and do not express behavior of objects
- Variables can be declared in the interfaces. They can only be declared as static and final. – Both keyword are assumed by default and can be safely omitted.
- Sometimes interfaces declare only constants – be used to effectively import sets of related constants.

Interface vs. Abstract Class

- An interface is simply a list of unimplemented, and therefore abstract, methods.
- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one super class.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Defining an Interface

- Defining an interface is similar to creating a new class.
- An interface definition has two components: the interface declaration and the interface body.

  interface Declaration {

    interface Body

  }

– The *interfaceDeclaration* declares various attributes about the interface such as its name and whether it extends another interface.

– The *interfaceBody* contains the constant and method declarations within the interface

```
public interface StockWatcher  {
    final String sunTicker = "SUNW";
    final String oracleTicker = "ORCL";
    final String ciscoTicker = "CSCO";
    void valueChanged
     (String tickerSymbol,
     double newValue);
}
```

If you do not specify that your interface is public, your interface will be accessible only to classes that are defined in the same package as the interface

Implementing an Interface

- Include an ‘implements’ clause in the class declaration.
- A class can implement more than one interface (the Java platform supports multiple inheritance for interfaces), so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.
- When implement an interface, either the class must implement all the methods declared in the interface and its superinterfaces, or the class must be declared abstract

```
class C {
    public static final int A = 1;
}
interface I {
    public int A = 2;
}
class X implements I {
    …
    public static void main (String[] args) {
     int  i = C.A, j = A; // 1 and 2 into i and j
     …
    }
}
```

```
public class StockMonitor
    implements StockWatcher {
  ...
  public void valueChanged
   (String tickerSymbol, double newValue) {
   if (tickerSymbol.equals(sunTicker))
      { ... }
   else if (tickerSymbol.equals(oracleTicker))
            { ... }
   else if (tickerSymbol.equals(ciscoTicker))
            { ... }
  }
}
```

Properties of Interface

- A new interface is a new reference data type.
- Interfaces are not instantiated with *new,* but they have certain properties similar to ordinary classes
- You can declare that an object variable will be of that interface type

e.g.

```
Comparable x = new Tile(…);
Tile y = new Tile(…);
if (x.compareTo(y) < 0) …
```

Superinterface (1)

- An interface can extend other interfaces, just as a class can extend or subclass another class.
- An interface can extend any number of interfaces.
- The list of superinterfaces is a comma-separated list of all the interfaces extended by the new interface

```
public interfaceName
 extends superInterfaces {
 InterfaceBody
}
```

Superinterface (2)

- Two ways of extending interfaces:
  - Add new methods
  - Define new constants
- Interfaces do not have a single top-level interface. (Like Object class for classes)

Interfaces Cannot Grow!

- Add some functionality to StockWatcher?

```
public interface StockWatcher
{
  final String sunTicker = "SUNW";
  final String oracleTicker = "ORCL";          final
String ciscoTicker = "CSCO";
  void valueChanged(String tickerSymbol,
        double newValue);
  void currentValue(String tickerSymbol,
        double newValue);
}
```

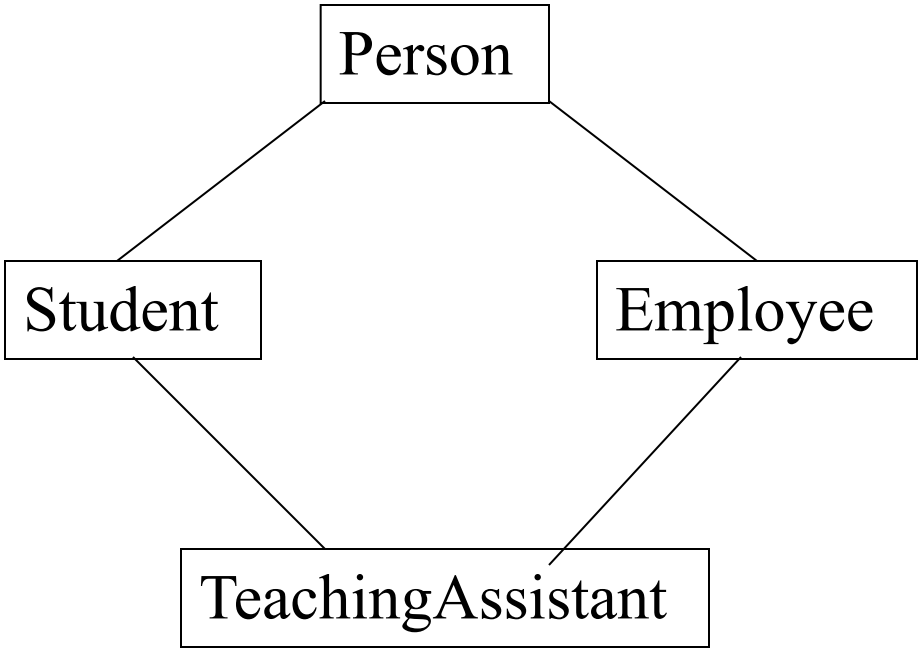
- If you make this change, all classes that implement the old interface will break because they don't implement the interface anymore!

- Try to anticipate all uses for your interface up front and specify it completely from the beginning.
- Create a StockWatcher subinterface called StockTracker that declared the new method:

```
public interface StockTracker
  extends StockWatcher  {
  void currentValue(String tickerSymbol,
        double newValue);
}
```

- Users can choose to upgrade to the new interface or to stick with the old interface.

Multiple Inheritance



- Multiple inheritance of class is not allowed in Java

```
class TeachingAssistant extends
    Student
{
    private EmployeeAttributes ea;
    …
    public String getEmployeeID()
    {
     return this.ea.getEmployeeID();
    }
}
```

EmployeeAttributes – non-public utility class

```
Interface Comparable
{
   int compareTo(Object o);
}

Class Student extends Person
     implements comparable
{
   parivate int SN; //Student number
   …
   public int compareTo(Object o)
   {
    return this.SN – ((Student)o).SN;
   }
   …
}
```

```
interface x
{
 char A = ‘A’;
 void gogi();
}
interface Y
{
 char B = ‘B’;
}
interface Z extends X, Y
{
 void dogi();
}
```

?What is in interface Z

Name conflicts in extending interfaces

- A class automatically implements all interfaces that re implemented by its superclass
- Interfaces belong to Java namespace and as such are placed into packages just like classes.
- Some interfaces are declared with entirely empty bodies. They serve as labels for classes
  - The most common marker interfaces are Cloneable and Serializable

```
class Car implement Cloneable
{ …
  public Object clone()
  {     return super.clone();
  }
}
```

```
interface X
{   char A = ‘A’;
    void gogi();
    void dogi();
}
interface Y
{   char A = ‘B’;
    void gogi(); //but not int gogi()
}
interface Z extends X, Y
{   void dogi(int i);
}
class C implements Z
{   public void gogi()
    {       char c = Y.A; …
    }
    public void dogi()
    {       char c = X.A; …
    }
    public void dogi(int I)
    {       …
     this.dogi(i – 1);
    }
}
```

What is Package?

- A *package* is a collection of related classes and interfaces providing access protection and namespace management.
  - To make classes easier to find and to use
  - To avoid naming conflicts
  - To control access

Why Using Packages?

- Programmers can easily determine that these classes and interfaces are related.
- Programmers know where to find classes and interfaces that provide graphics-related functions.
- The names of classes won't conflict with class names in other packages, because the package creates a new namespace.
- Allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package

Put Your Classes and Interfaces into Packages

- It is no need to “create” packages, they come to existence as soon as they are declared
- The first line of your source file:

  package <package_name>;

  e.g. package cars;

  class Car

  { … }
- Every class must belong to one package.
- Only one package statement is allowed per source file.
- If package statement is omitted, the class belongs to a special default package – the only package in Java that has no name.

Subpackages

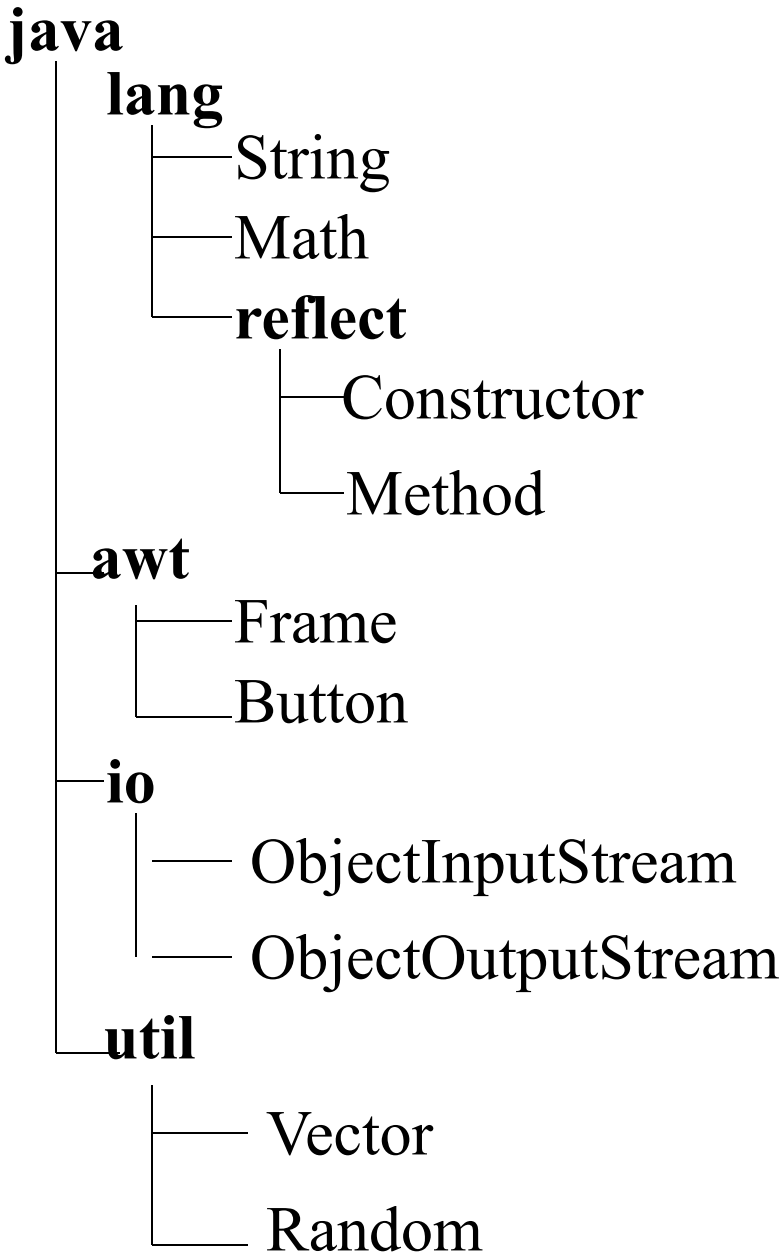
- We can create hierarchies of nested packages

  e.g.

  package machines.vehicles.cars;
  class FastCar extends Car
  {…}

- Subpackage names must always be prefixed with names of all enclosing packages.
- Package hierarchy does not have a single top-level all-encompassing root package. Instead, we deal with a collection of top-level packages

Partial Package Tree of Java



Packages and Class

- Packages subdivide name space of Java classes

```
machines.vehicles.cars.FastCar mycar =
        new machine.vehicles.cars.FastCar();
```

- Package names should be made as unique as possible to prevent name clashes
- Class name must be fully qualified with their package name. Except:
  - Class name immediately following the class keyword, e.g. class Car{…}
  - Class belongs to the same package as the class being currently defined
  - Class is explicitly specified in one of the import statements
  - Class belongs to java.lang package (all java.lang classes are implicitly imported)

import Statement

- To avoid typing fully qualified names – use import statements sandwiched between the package statement and class definition.

```
e.g.  package grand.prix;
      import java.lang.*;     //implicitly specified
      import java.util.Random;
      import machines.vehicles.cars.*;
      import machines.*.*; //COMPILER ERROR!
      import Racer;     //from default package

      class SuperFastCar extends FastCar
      {
        private Racer r;
        public static void main(String{[] args)
        {
              Random RNG = new Random();
              java.util.Vector v =
                    new java.util.Vector();
        }
      }
```

Store Class Files

- Class files must be stored in a directory structure that mirrors package hierarchy

  e.g. Both .java and .class files for FastCar class from the machines.vehicles.cars package can be stored in the following directory:

C:\A\B\classes\machines\vehicles\cars

Classpath

- Compiled classes can be stored in different locations in the file systems.
- How to locating these files – Setting a classpath which is a list of directories where class files should be searched

  e.g. If Java system is to find classes from the machines.vehicles.cars package , its classpath must point to C:\A\B\classes, the root of the package directory hierarchy.

Setup Classpath

- In command line

  c:\> java –classpath C:\A\B\classes machines.vehicles.cars.Car

- To avoid specify the classpath in every command, set classpath as a command shell environment variable:
  - Windows:

  >set CLASSPATH = C:\A\B\classes
  >set CLASSPATH = .; %CLASSPATH%
  >echo %CLASSPATH%
  .; C:\A\B\classes

  - Unix:

  $CLASSPATH = $HOME/A/B/classes
  $CLASSPATH = .: $CLASSPATH
  $echo $CLASSPATH
  .: home/santa/A/B/classes