CHAPTER 9

INHERITANCE AND INTERFACES

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
10/29/2011

Comparing Interfaces and Inheritance

- Here is a different interface: `Named`

```
public interface Named
{
   String getName();
}
```

- A class can implement more than one interface:

```
public class Country implements Measurable, Named
```

- A class can only extend (inherit from) a single superclass.
- An interface specifies the behavior that an implementing class should supply - no implementation.
- A superclass provides some implementation that a subclass inherits.
- Develop interfaces when you have code that processes objects of different classes in a common way.

Casting from Interfaces to Classes

- Method to return the object with the largest measure:

```
public static Measurable larger(Measurable obj1,
   Measurable obj)
{
   if (obj1.getMeasure() > obj2.getMeasure())
   {
      return obj1;
   }
   else
   {
      return obj2;
   }
}
```

- Returns the object with the larger measure, as a `Measurable` reference.

```
Country uruguay = new Country("Uruguay", 176220);
Country thailand = new Country("Thailand", 513120);
Measurable max = larger(uruguay, thailand);
```

Casting from Interfaces to Classes

- You know that `max` refers to a `Country` object, but the compiler does not.
- Solution: cast

```
Country maxCountry = (Country) max;
String name = maxCountry.getName();
```

- You need a cast to convert from an interface type to a class type.
- If you are wrong and `max` doesn't refer to a `Country` object, the program throws an exception at runtime.
- If a `Person` object is actually a `Superhero`, you need a cast before you can apply any `Superhero` methods.

The Comparable Interface

- Comparable interface is in the standard Java library.
- Comparable interface has a single method:

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

- The call to the method:

```
a.compareTo(b)
```

- The compareTo method returns:
  - a negative number if a should come before b,
  - zero if a and b are the same
  - a positive number if b should come before a.
- Implement the Comparable interface so that objects of your class can be compared, for example, in a sort method.

The Comparable Interface

- BankAccount class' implementation of Comparable:

```
public class BankAccount implements Comparable
{
   . . .
   public int compareTo(Object otherObject)
   {
      BankAccount other = (BankAccount) otherObject;
      if (balance < other.balance) { return -1; }
      if (balance > other.balance) { return 1; }
      return 0;
   }
   . . .
}
```

- compareTo method has a parameter of reference type Object
- To get a BankAccount reference:

```
BankAccount other = (BankAccount) otherObject;
```

The Comparable Interface

- Because the `BankAccount` class implements the `Comparable` interface, you can sort an array of bank accounts with the `Arrays.sort` method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

- Now the accounts array is sorted by increasing balance.
- The `compareTo` method checks whether another object is larger or smaller.

Self Check

Write a method `max` that finds the larger of any two `Comparable` objects.

Answer:

```
public static Comparable max(Comparable a,
   Comparable b)
{
   if (a.compareTo(b) > 0) { return a; }
   else { return b; }
}
```

Inner Classes

- Trivial class can be declared inside a method:

```
public class MeasurerTester
{
   public static void main(String[] args)
   {
      class AreaMeasurer implements Measurer
      {
         . . .
      }
      . . .
      Measurer areaMeas = new AreaMeasurer();
      double averageArea = Data.average(rects,
  areaMeas);
      . . .
   }
}
```

Inner Classes

- You can declare inner class inside an enclosing class, but outside its methods.
- It is available to all methods of enclosing class:

```
public class MeasurerTester
{
   class AreaMeasurer implements Measurer
   {
      . . .
   }
   public static void main(String[] args)
   {
      Measurer areaMeas = new AreaMeasurer();
      double averageArea = Data.average(rects, areaMeas);
      . . .
   }
}
```

Inner Classes

- Compiler turns an inner class into a regular class file with a strange name:

```
MeasurerTester$1AreaMeasurer.class
```

- Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.

Implementing Interfaces

- An interface can be implemented by multiple classes.
- Each implementing class can provide their own unique versions of the method definitions.

```
interface I1 {
  void m1() ;
}

class C1 implements I1 {
  public void m1() { System.out.println(“Implementation in C1”); }
}

class C2 implements I1 {
  public void m1() { System.out.println(“Implementation in C2”); }
}
```

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
{
   // all methods of both interfaces
}
```

Implementing More Than One Interface

```
interface I1 {
   void m1();
}
interface I2 {
   void m2() ;
   void m3() ;
}
class C implements I1, I2 {
   public void m1() { System.out.println("C-m1"); }
   public void m2() { System.out.println("C-m2"); }
   public void m3() { System.out.println("C-m3"); }
}
```

C must implement all methods in I1 and I2.

Resolving Name Conflicts Among Interfaces

- Since a class may implement more than one interface, the names in those interfaces may collide.
- To solve name collisions, Java use a simple mechanism.
- Two methods that have the same name will be treated as follows in Java:
  - If they are different signature, they are considered to be overloaded.
  - If they have the same signature and the same return type, they are considered to be the same method and they collapse into one.
  - If they have the same signature and the different return types, a compilation error will occur.

Resolving Name Conflicts Among Interfaces

```
interface I1 {
   void m1();
   void m2();
   void m3();
}
interface I2 {
   void m1(int a);
   void m2();
   int m3();
}
class C implements I1, I2 {
   public void m1() { … }          // implementation of m1 in I1
   public void m1(int x) { … }     // implementation of m1 in I2
   public void m2() { … }          // implementation of m2 in I1 and I2
}
```

There will be a compilation error for m3.

Inheritance Relation Among Interfaces

- Same as classes, interfaces can hold inheritance relation among them

```
interface I2 extends I1 { … }
```

- Now, I2 contains all abstract methods of I1 plus its own abstract methods.
- The classes implementing I2 must implement all methods in I1 and I2.

Interfaces as Data Types

- Interfaces (same as classes) can be used as data types.
- Different from classes: We cannot create an instance of an interface.

```
interface I1 { … }
class C1 implements I1 { … }
class C2 extends C1 { … }

// a variable can be declared as type I1
I1 x;
```

- A variable declared as I1, can store objects of C1 and C2.
  - More later…

The Iterator Interface

- As we discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
- The `hasNext` method returns a boolean result – true if there are items left to process
- The `next` method returns the next object in the iteration
- The `remove` method removes the object most recently returned by the `next` method

The Iterator Interface

- By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators
- The programmer must decide how best to implement the iterator functions
- Once established, the for-each version of the `for` loop can be used to process the items in the iterator

Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java

When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses are expected to fulfill a similar role in different ways through different implementations
  - These subclasses extend the same Abstract class and provide different implementations for the abstract methods
- Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Why do we use Interfaces? Reason #1

- 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

Why do we use Interfaces? Reason #2

- 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

Why do we use Interfaces? Reason #3

- To model multiple inheritance
  - A class can implement multiple interfaces while it can extend only one class

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

Problem of Rewriting an Existing Interface

- Consider an interface that you have developed called DoIt:

```
public interface DoIt {
   void doSomething
   int doSomethingE
}
```

- Suppose that, add a third me interface now

```
public interface DoIt
   void doSomething
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);
}
```

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

Solution of Rewriting an Existing Interface

- Create more interfaces later
- For example, you could create a DoItPlus interface that extends DoIt:

```
public interface DoItPlus extends DoIt {
   boolean didItWork(int i, double x, String s);
}
```

- Now users of your code can choose to continue to use the old interface or to upgrade to the new interface

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 sub-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