Abstract Classes and Interfaces

Abstract Classes

- An abstract class in a class hierarchy represents a generic concept
  - Common elements in a hierarchy that are too generic to instantiate
  - Cannot be instantiated
- abstract on the class header:

```
public abstract class Product
{
    // contents
}
```

Abstract Classes

- abstract classes typically have:
  - abstract methods with no definitions (like an interface)
  - probably also non-abstract methods with full definitions
- Does not have to contain abstract methods -- simply declaring it as abstract makes it so

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract

Abstract Classes

```
abstract class A1 {
        abstract void m1();
        abstract String m2();
}

class C1 extends A1 {
        void m1() { System.out.println("C1-m1"); }
        String m2() { return "C1-m2"; }
}

abstract C2 extends A1 {
        void m1() { System.out.println("C2-m1"); }
}
```

➔ C2 must be abstract, because it does not implement the abstract method m2.

Abstract Classes

- Abstract methods cannot be defined as final or static
  - final cannot be overridden (contradiction!)
  - static could be invoked by just using the name of the class – can't invoke it with no implementation

Interfaces

- A Java interface is a collection of abstract methods and constants
  - An abstract method is a method header without a method body
  - abstract - but because all methods in an interface are abstract, usually it is left off
- An interface establishes a set of methods that a class will implement
  - Similar to abstract class but all methods are abstract (and all properties are constant)

Interfaces

```
interface I1 {
        int CONST1=5;

        void m1();
}
```

Although we do not write here, it is assumed that CONST1 is declared as a constant (with keywords public, final and static)

Although we do not write here, it is assumed that m1 is declared with keywords public and abstract.

Interface methods are public by default

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

- Defines similarities that multiple classes share
  - to tie elements of several classes together - without having an inheritance relationship, so still no multiple inheritance
  - separate design from coding
- An interface **cannot be instantiated**
- A class implements an interface by:
  - stating so in the class header
  - Implementing all abstract methods in the interface, plus maybe some others

Interfaces

```
public class CanDo 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

Implementing Interfaces (cont.)

- 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() ;    C must implement all methods in I1 and I2.
        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"); }
}
```

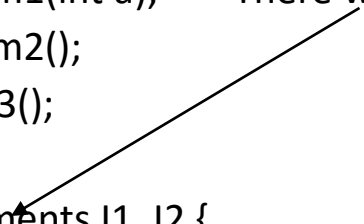
Resolving Name Conflicts Among Interfaces

- 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…

Interfaces

- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants
- A class that implements an interface can implement other methods as well
- See Complexity.java
- See Question.java
- See MiniQuiz.java

```java
//********************************************************************
//  Complexity.java       Author: Lewis/Loftus
//
//  Represents the interface for an object that can be assigned an
//  explicit complexity.
//********************************************************************

public interface Complexity
{
   public void setComplexity (int complexity);
   public int getComplexity();
}
```

```
//********************************************************************
//  Question.java       Author: Lewis/Loftus
//
//  Represents a question (and its answer).
//********************************************************************

public class Question implements Complexity
{
   private String question, answer;
   private int complexityLevel;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the question with a default complexity.
   //-----------------------------------------------------------------
   public Question (String query, String result)
   {
      question = query;
      answer = result;
      complexityLevel = 1;
   }
```

continue

continue

```
//-----------------------------------------------------------------
//  Sets the complexity level for this question.
//-----------------------------------------------------------------
public void setComplexity (int level)
{
   complexityLevel = level;
}

//-----------------------------------------------------------------
//  Returns the complexity level for this question.
//-----------------------------------------------------------------
public int getComplexity()
{
   return complexityLevel;
}

//-----------------------------------------------------------------
//  Returns the question.
//-----------------------------------------------------------------
public String getQuestion()
{
   return question;
}
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Returns the answer to this question.
    //-----------------------------------------------------------------
    public String getAnswer()
    {
       return answer;
    }

    //-----------------------------------------------------------------
    //  Returns true if the candidate answer matches the answer.
    //-----------------------------------------------------------------
    public boolean answerCorrect (String candidateAnswer)
    {
       return answer.equals(candidateAnswer);
    }

    //-----------------------------------------------------------------
    //  Returns this question (and its answer) as a string.
    //-----------------------------------------------------------------
    public String toString()
    {
       return question + "\n" + answer;
    }
}
```

```
//********************************************************************
//  MiniQuiz.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a class that implements an interface.
//********************************************************************

import java.util.Scanner;

public class MiniQuiz
{
   //-----------------------------------------------------------------
   //  Presents a short quiz.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Question q1, q2;
      String possible;

      Scanner scan = new Scanner (System.in);

      q1 = new Question ("What is the capital of Jamaica?",
                         "Kingston");
      q1.setComplexity (4);

      q2 = new Question ("Which is worse, ignorance or apathy?",
                         "I don't know and I don't care");
      q2.setComplexity (10);
```

continue

continue

```
        System.out.print (q1.getQuestion());
        System.out.println (" (Level: " + q1.getComplexity() + ")");
        possible = scan.nextLine();
        if (q1.answerCorrect(possible))
           System.out.println ("Correct");
        else
           System.out.println ("No, the answer is " + q1.getAnswer());

        System.out.println();
        System.out.print (q2.getQuestion());
        System.out.println (" (Level: " + q2.getComplexity() + ")");
        possible = scan.nextLine();
        if (q2.answerCorrect(possible))
           System.out.println ("Correct");
        else
           System.out.println ("No, the answer is " + q2.getAnswer());
    }
}
```

continu

Sample Run

```
What is the capital of Jamaica? (Level: 4)
Kingston
Correct

Which is worse, ignorance or apathy? (Level: 10)
apathy
No, the answer is I don't know and I don't care
```

```
                                                                    ;
                                                                    ());
        System.out.println();
        System.out.print (q2.getQuestion());
        System.out.println (" (Level: " + q2.getComplexity() + ")");
        possible = scan.nextLine();
        if (q2.answerCorrect(possible))
           System.out.println ("Correct");
        else
           System.out.println ("No, the answer is " + q2.getAnswer());
    }
}
```

Example

- Example: a method to compute the average of an array of `Object`s
  - The algorithm for computing the average is the same in all cases
  - Details of measurement differ
- Goal: write one method that provides this service.
- We can't call `getBalance` in one case and `getArea` in another.
- Solution: all object who want this service must agree on a `getMeasure` method
  - `BankAccount`'s `getMeasure` will return the balance
  - `Country`'s `getMeasure` will return the area
- Now we implement a single `average` method that computes the sum:

```
sum = sum + obj.getMeasure();
```

Defining an Interface Type

- Problem: we need to declare a type for `obj`
- Need to invent a new type that describes any class whose objects can be measured.
- An interface type is used to specify required operations (like `getMeasure`):

```
public interface Measurable
{
   double getMeasure();
}
```

- A Java interface type declares methods but does not provide their implementations.

Syntax 8.1 Declaring an Interface

Syntax

```
public interface InterfaceName
{
   method headers
}
```

```
public interface Measurable
{
   double getMeasure();
}
```

The methods of an interface are automatically public.

No implementation is provided.

Defining an Interface Type

- Implementing a reusable `average` method:

```
public static double average(Measurable[] objects)
{
   double sum = 0;
   for (Measurable obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   if (objects.length > 0) { return sum / objects.length; }
   else { return 0; }
}
```

- This method is can be used for objects of any class that conforms to the `Measurable` type.

© gregory horler/iStockphoto.

Implementing an Interface Type

- Use `implements` reserved word to indicate that a class implements an interface type:

```
public class BankAccount implements Measurable
{
   …
   public double getMeasure()
   {
   return balance;
   }
}
```

- `BankAccount` objects are instances of the `Measurable` type:

```
Measurable obj = new BankAccount(); // OK
```

Implementing an Interface Type

- A variable of type `Measurable` holds a reference to an object of some class that implements the `Measurable` interface.
- `Country` class can also implement the `Measurable` interface:

```
public class Country implements Measurable
{
   public double getMeasure()
   {
      return area;
   }
   . . .
}
```

- Use interface types to make code more reusable.

Implementing an Interface Type

- Put the `average` method in a class - say `Data`

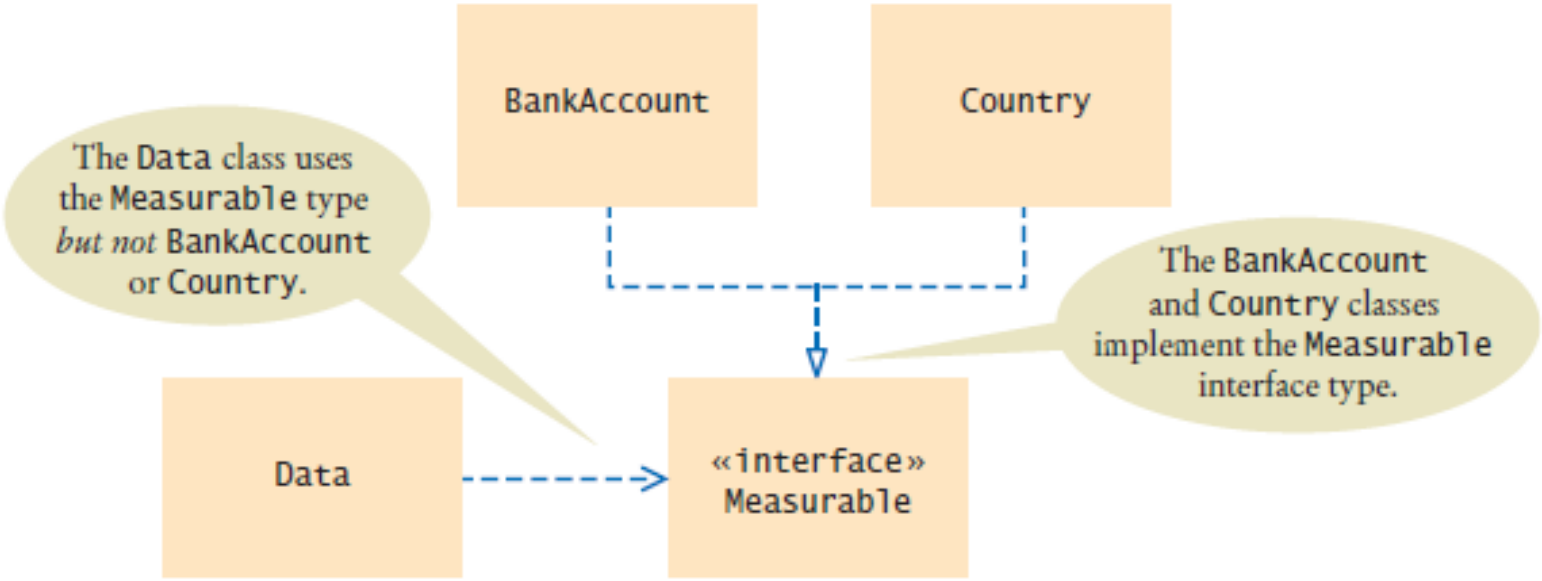


Figure 1 UML Diagram of the `Data` Class and the Classes that Implement the `Measurable` Interface

- `Data` class is decoupled from the `BankAccount` and `Country` classes.

section_1/Data.java

```
1  public class Data
2  {
3     /**
4        Computes the average of the measures of the given objects.
5        @param objects an array of Measurable objects
6        @return the average of the measures
7     */
8     public static double average(Measurable[] objects)
9     {
10       double sum = 0;
11       for (Measurable obj : objects)
12       {
13          sum = sum + obj.getMeasure();
14       }
15       if (objects.length > 0) { return sum / objects.length; }
16       else { return 0; }
17    }
18 }
```

section_1/MeasurableTester.java

```
1  /**
2     This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasurableTester
5  {
6     public static void main(String[] args)
7     {
8        Measurable[] accounts = new Measurable[3];
9        accounts[0] = new BankAccount(0);
10       accounts[1] = new BankAccount(10000);
11       accounts[2] = new BankAccount(2000);
12
13       double averageBalance = Data.average(accounts);
14       System.out.println("Average balance: " + averageBalance);
15       System.out.println("Expected: 4000");
16
17       Measurable[] countries = new Measurable[3];
18       countries[0] = new Country("Uruguay", 176220);
19       countries[1] = new Country("Thailand", 513120);
20       countries[2] = new Country("Belgium", 30510);
21
22       double averageArea = Data.average(countries);
23       System.out.println("Average area: " + averageArea);
24       System.out.println("Expected: 239950");
25    }
26 }
```

Continued

section_1/MeasurableTester.java

Program Run:

```
Average balance: 4000
Expected: 4000
Average area: 239950
Expected: 239950
```

Self Check 8.1

Suppose you want to use the `average` method to find the average salary of an array of `Employee` objects. What condition must the `Employee` class fulfill?

Answer: It must implement the `Measurable` interface, and its `getMeasure` method must return the salary.

Self Check 8.4

What is wrong with this code?

```
Measurable meas = new Measurable();
System.out.println(meas.getMeasure());
```

Answer: `Measurable` is not a class. You cannot construct objects of type `Measurable`.

Self Check 8.5

What is wrong with this code?

```
Measurable meas = new Country("Uruguay", 176220);
System.out.println(meas.getName());
```

Answer: The variable `meas` is of type `Measurable`, and that type has no `getName` method.

Interfaces

- The Java standard class library contains many helpful interfaces
- The **Comparable interface** contains one abstract method called compareTo, which is used to compare two objects
- We discussed the **compareTo** method of the **String class** before
- The String class implements Comparable, giving us the ability to put strings in lexicographic order

The Comparable Interface

- Any class can implement Comparable to provide a mechanism for comparing objects of that type

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

  - **The value returned from `compareTo` should be negative if `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`**
  - **When a programmer designs a class that implements the `Comparable` interface, it should follow this intent**

The Comparable Interface

- It's up to the programmer to determine what makes one object less than another
- For example, you may define the compareTo method of an Employee class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

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.

© Janis Dreosti/iStockphoto.

Self Check 8.14

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; }
}
```

Self Check 8.15

Write a call to the method of Self Check 14 that computes the larger of two bank accounts, then prints its balance.

Answer:

```
BankAccount larger =
   (BankAccount) max(first, second);
System.out.println(larger.getBalance());
```

Note that the result must be cast from `Comparable` to `BankAccount` so that you can invoke the `getBalance` method.

The Iterator Interface

- As we discussed, 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

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 i, double x);
        int doSomethingElse(String s);
}
```

- Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

```
public interface DoIt {
        void doSomething(int i, double x);
        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