CSC220
Chapter 1: Object-Oriented Programming and Class Hierarchies

Mitsunori Ogihara

University of Miami

August 31, 2010

Outline

Objectives

1. Abstract data types (ADT)
2. Interfaces and their roles
3. Inheritance and its fascilitation
4. Method selection at runtime
5. Abstract classes
6. Exception hierarchy
7. Class `Object` and method overriding
8. Package creation and package visibility.

ADT

Abstract Data Type (ADT for short): a central concept of Java. It consists of

- data encapsulated in the type and
- methods executed on the data.

Abstract data types enable their users to write applications without knowing details of how data are stored and how methods are implemented.

Interfaces

A Java interface is a way to specify or describe an ADT to an applications programmer.
It is like a contract that an applications programmer must fulfill.
An interface may specify:

- a required set of methods, each along with:
  - the types of input arguments, if any, and
  - the type of return object, if any;
- constants.

Interface Format

The format for an interface is

```
public interface InterfaceName {
   abstract method headings
   constant declarations
}
```

Things to note about the attribute:

1. The method header has the attribute of **public abstract** and thus can be accompnied with any of its sub-word.
2. The constant header has the attribute of **public static final** and thus can be accompanied with any of its sub-word.

Example

```
public interface OneNameInt {
    /* Default name to be used */
    static final String DEFAULT_FIRST_NAME = "Troy";
    /**
     * Set a new value to the name
     * @param s the new name
     */
    void setName(String s);
    /**
     * Obtain the name as a string
     * @return a string representation of the name
     */
    String getName();
}
```

Properties of Interfaces

- An interface specifies only contract and thus fulfillment of the contract is up to the programmer that **implements** it.
- An interface can be used as type of an object, but since it only specifies contract you can for constructing an object.

Example

An interface can be extended by another interface.

```
public interface TwoNamesInt extends OneNameInt {
    /**
     * set the other name
     * @param s the other name
     */
    abstract public void setOtherName(String s);
    /**
     * Obtain the other name
     * @return the other name
     */
    abstract public String getOtherName();
}
```

These methods are *additions*; those defined in OneNameInt are **inherited** by the new interface.

Implementing an Interface

A class that fulfills contract of an interface is said to **implement** the interface.
With the keyword of `implements` header specifies this relation:

```
class C implements I
```

states that the class C is one that implements interface I.

Instantiation of Class Implementing an Interface

The object should be created with a constructor of the class.
The type of the object can be:

- either the interface or one of its super-interfaces (if any); or
- either the class itself or one of its superclasses.

In the previous example the following are valid declarations:

```
PersonNames a = new PersonNames("Jean Thielemans", "Toots");
Object b = new PersonNames("Milt Jackson", "Bags");
OneNameInt c = new PersonNames("Albert Heath", "Tootie");
TwoNamesInt d = new PersonNames("Jeff Watts", "Tain");
```

The phenomenon in which the same instance can be treated as of multiple types is called **polymorphism**.

Abstract Class

Abstract class is a type residing between interface and class.

- Abstract class may contain abstract methods.
- Abstract class may have constructors; however, it cannot be instantiated; the constructors can be used only in constructors of its immediate subclasses.
- Abstract class may define data fields.

Example

```
public abstract class University implements TwoNamesInt {
   private String nameOne;
   private String nameTwo;
   University() {
      nameOne = "University of Miami";
      nameTwo = "UM";
   }
   /**
    * obtain population
    * @return an integer value
    */
   public abstract int getPopulation();
   public void setName(String s) {
      this.nameOne = s;
   }
   public String getName() {
      return this.nameOne;
   }
   public void setOtherName(String s) {
      this.nameTwo = s;
   }
   public String getOtherName() {
      return this.nameTwo;
   }
}
```

Use of Abstract Class

- An abstract class may have a constructor, which can be invoked from a class that extends it.
- An abstract class can be used as type, but cannot be instantiated, so constructors of an abstract class can be used only in constructors of its extensions.

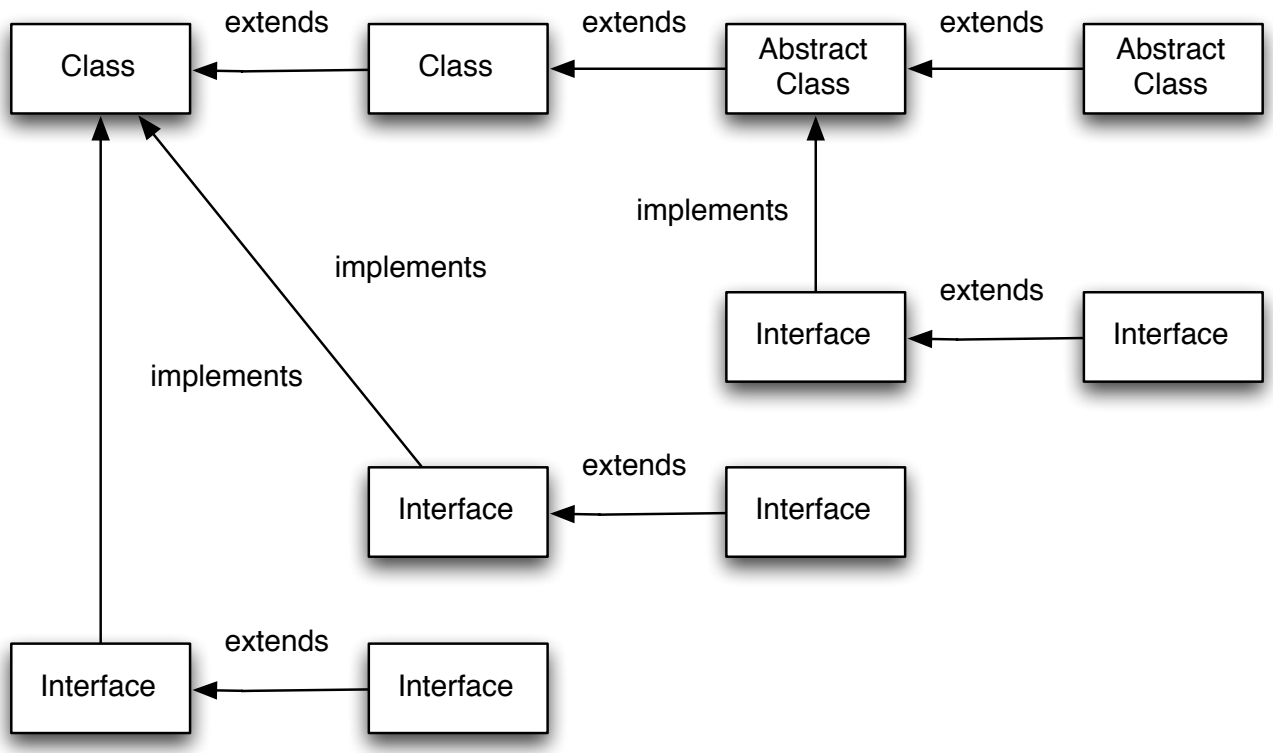
Example

```
public class PrivateUniversity extends University {
   private int pop;
   private String stateInitial;
   PrivateUniversity() {
      super(); // parent's constructor
      super.setOtherName("The U");
      this.pop = 100000;
      this.stateInitial = "FL";
   }
   PrivateUniversity(String s1, String s2, int n, String s3) {
      this.pop = n;
      this.setName(s1);
      super.setName(s2);
      this.stateInitial = s3;
   }
   @Override
   public int getPopulation() {
      return this.pop;
   }
   /**
    * Obtain the state initial
    * @return a string object
    */
   public String getStateInitial() {
      return this.stateInitial;
   }
}
```

Use of Keywords “this” and “super”

- `this.methodName(...)` in the code of class C refers to the method `methodName` defined in the class.
- `this.dataName` in the code of class C refers to the data field `dataName` within the class. Without `this.`, it refers to something that is locally obvious to the program; e.g., data by that name given as an argument of a method in which the reference appears.
- `super.methodName(...)` in the code of class C that extends a class D refers to the method `methodName` defined in D.
- `super.dataName` in the code of class C that extends a class D refers to the data field `dataName` defined in D.

Extension and Implementation



Method Overloading and Selection at Runtime

A class may have multiple methods with the same, as long as no two of the methods with the same name have the same list of arguments.

For example, in the previous example of PrivateUniversity, having

```
public boolean changeInfo(String a)
public int changeInfo(String a, String b)
public void changeInfo(String a, String b, String c)
```

for respectively changing information of

- the real name,
- the real name and the nick name,
- the real name, the nick name and the state initials

is legitimate. Note that they may have different return types (int, boolean, and void).

Empty Constructor as Default

- An empty constructor is a constructor with no arguments; i.e., a constructor of the form `ClassName()`.
- If a class has nonempty constructor, its empty constructor must be explicitly defined.

Class Object

Every non-primitive data type is an extension of Object.

Objects has, in addition to a few others, five important public methods:

- `Object clone()` for obtaining a copy of an object;
- `boolean equals(Object obj)` for reference equality; `x.equals(y)` tests whether `x` and `y` refer to the same object;
- `Class getClass()` for class reference;
- `int hashCode()` for obtaining an integer assigned to an object;
- `String toString()` for obtaining a String representation of an object

Clone

The method `clone` defined in texttt{Object} is a **shallow copy** method, in the sense that the method copies only primitive data types.

As opposed to this, a clone method that copies all the data fields is called a **deep copy** method.

The clone method of Object is not visible to its subclasses.

Equals

The code

```
PrivateUniversity p1 =
   new PrivateUniversity("U. Miami", "UM", 10000, "FL");
PrivateUniversity p2 =
   new PrivateUniversity("U. Miami", "UM", 10000, "FL");
if (p1.equals(p2)) System.out.println("p1 equals p2");
else System.out.println("p1 does not equal p2");
```

produces the second output.

getClass, hashCode, toString

- getClass gives the class information. `x.getClass().getName()` returns the string object that is the class name of the class of x.
- x.hashCode() can be used when building hash tables.
- x.toString() returns a string encodig of x.

Overriding

- A class can redefine any methods that are defined in the class it extends.
- The Object methods `clone` (*which is not accessible to classes that extend Object*) and `toString` are those that may need to be redefined.
- **Method override** refers to redefining methods provided in a super-class.
- For Javadoc use keyword `@override` to indicate method override.
- To access the unoverridden method, attach `super.`

Abstract Class Number

An abstract class that provides the basis for an object version of the primitive number data types (byte, double, float, int, long, and short).

The Object versions are: Byte, Double, Float, Integer, Long, and Short.

Conversion Between a Number Class and Primitive Version

For only single-object reference, Java automatically converts between the primitive data type version and the Number sub-class version.

```
Integer A = new Integer(10);
int a = A;
Integer B = a;
```

is a valid code.

This doesn't work for arrays; i.e., Java won't convert between int[] and Integer[].

Casting

If an object *O* is of class *C* and *C* extends *D*, then *O* can be treated as an object of class *C* as well as an object of class *D*. Suppose Class Bar extends Class Foo.

- Suppose variable x is of type Foo. You can execute: `x = new Bar(...)` and then `((Bar)x).someMethod` to apply `someMethod` on x being treated as a Bar object.
- Suppose y is of type Bar. Suppose both classes have their own toString() method. Then you can execute `((Foo)y).toString()` to execute the Foo-class method on y.

The former is **down-casting** and the latter is **up-casting**.

instanceof

The **instanceof** allows you to test whether an object is of certain class.
In the previous example, x is not only an instance of Foo but also of a Bar, so

- `(x instanceof Foo)` and
- `(x instanceof Bar)`

evaluate to `true`

Class Class

Class is a special class that deals with classes.
`x.getClass()` method provides the class of object x.