Object Oriented Programming with Java

CHAPTER 9, 10 (INTERFACES)

Creating and Using Abstract Classes

- **Abstract class**
  - Cannot create any concrete objects
  - Can inherit
  - Usually has one or more empty abstract methods
- When declaring an abstract class:
  - Use the keyword `abstract`
  - Provide the superclass from which other objects can inherit
  - Example:

```
public abstract class Animal
```

Creating and Using Abstract Classes (cont'd.)

- An **abstract method** does not have:
  - A body
  - Curly braces
  - Method statements
- To create an abstract method:
  - Use the keyword `abstract`
  - The header must include the method type, name, and parameters
  - Include a semicolon at the end of the declaration
  - Example:

```
public abstract void speak();
```

Creating and Using Abstract Classes (cont'd.)

- Subclass of abstract class
  - Inherits the abstract method from its parent
    - Must provide the implementation for the inherited method or be abstract itself

Using Dynamic Method Binding

- Every subclass object "is a" superclass member
  - Convert subclass objects to superclass objects
  - Can create a reference to a superclass object
    - Create a variable name to hold the memory address
    - Store a concrete subclass object
    - Example:

```
Animal animalRef;
animalRef = new Cow();
```

Using Dynamic Method Binding (cont'd.)

- **Dynamic method binding**
  - Also called **late method binding**
  - An application's ability to select the correct subclass method
- When an application executes, the correct method is attached (or bound) to the application based on current and changing (dynamic) context

Using Dynamic Method Binding (cont'd.)

```
public class AnimalReference
{
   public static void main(String[] args)
   {
      Animal animalRef;
      animalRef = new Cow();
      animalRef.speak();
      animalRef = new Dog();
      animalRef.speak();
   }
}
```

Figure 11-8 The `AnimalReference` application

Using a Superclass as a Method Parameter Type

- Useful when you want to create a method that has one or more parameters that might be one of several types
- Use dynamic method binding

```
public static void talkingAnimal
  (Animal animal)
Dog dog = new Dog();
talkingAnimal(dog);
```

Using a Superclass as a Method Parameter Type (cont'd.)

```
public class TalkingAnimalDemo
{
   public static void main(String[] args)
   {
      Dog dog = new Dog();
      Cow cow = new Cow();
      dog.setName("Ginger");
      cow.setName("Molly");
      talkingAnimal(dog);
      talkingAnimal(cow);
   }
   public static void talkingAnimal(Animal animal)
   {
      System.out.println("Come one. Come all.");
      System.out.println
         ("See the amazing talking animal!");
      System.out.println(animal.getName() +
         " says");
      animal.speak();
      System.out.println("**************");
   }
}
```

Figure 11-10 The `TalkingAnimalDemo` class

Creating Arrays of Subclass Objects

- Create a superclass reference
  - Treat subclass objects as superclass objects
    - Create an array of different objects that share the same ancestry
- Create an array of three `Animal` references

```
Animal[] animalRef = new Animal[3];
```

  - Reserve memory for three `Animal` object references
    - Animal[0] = new Dog(“Snoopy”);
    - Animal[1] = new Cat(“Tom”);

  - We can do the same with an ArrayList object

Creating and Using Interfaces

- **Multiple inheritance**
  - Inherit from more than one class
  - Not available in Java
  - Variables and methods in parent classes might have identical names
    - Creates conflict
    - Which class should `super` refer to when a child class has multiple parents?

Creating and Using Interfaces (cont'd.)

- **Interface**
  - An alternative to multiple inheritance
  - Looks like a class except all of its methods are implicitly `public` and `abstract`
  - A description of what a class does
    - Declares method headers
  - Each Interfaces will be regarded as a parent (super class)

Creating and Using Interfaces (cont'd.)

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}
public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}
```

Figure 11-24 The `Animal` and `Dog` classes

Creating and Using Interfaces (cont'd.)

```
public interface Worker
{
   public void work();
}
```

Figure 11-25 The `Worker` interface

Creating and Using Interfaces (cont'd.)

```
public class WorkingDog extends Dog implements Worker
{
   private int hoursOfTraining;
   public void setHoursOfTraining(int hrs)
   {
      hoursOfTraining = hrs;
   }
   public int getHoursOfTraining()
   {
      return hoursOfTraining;
   }
   public void work()
   {
      speak();
      System.out.println("I am a dog who works");
      System.out.println("I have " + hoursOfTraining +
         " hours of professional training!");
   }
}
```

Figure 11-26 The `WorkingDog` class

Creating and Using Interfaces (cont'd.)

- Create an interface
  - Example:
    `public interface Worker`
- Implement an interface
  - Use the keyword `implements`
    - Requires the subclass to implement its own version of each method
  - Use the interface name in the class header
    - Requires class objects to include code

```
public class WorkingDog extends Dog
implements Worker
```

Class and Interface

- Abstract classes versus interfaces
  - You cannot instantiate concrete objects of either
    - i.e., can't use new: Animal xAnim = new Animal(); // N/A
  - Abstract classes
    - Can contain nonabstract methods
    - Provide data or methods that subclasses can inherit
      - Subclasses maintain the ability to override inherited methods
    - Can include methods that contain the actual behavior the object performs

Class and Interface (cont)

- Abstract classes versus interfaces (cont'd.)
  - Interfaces
    - Methods must be abstract
    - Programmers know what actions to include
    - Every implementing class defines the behavior that must occur when the method executes
    - A class can implement behavior from more than one parent

Class and Interface (cont)

- A class can only extend (inherit from) a single superclass.
  - A superclass provides some implementation that a subclass inherits.
- An interface specifies the behavior that an implementing class should supply
- Develop interfaces when you have code that processes objects of different classes in a common way.

- A class can implement more than one interfaces:
  - **public class** *Animal* **implements** Worker, Speaker
  - A class that does not implement the methods specified in the interface must be declared to be an abstract class

Class and Interface (cont)

- Interface are treated as a parent class
- It is possible to create an object of the interface and store the subclass object in it
  - Speaker animalSpeaker = **new Lion**("Simba", "Roar");
  - animalSpeaker.speak();
    - animalSpeaker.work(); // not possible
- We need a cast to convert from an interface type to a class type.
  - *Lion* lionWorker = (*Lion*) animalSpeaker;
  - lionWorker.**work**();

Creating Interfaces to Store Related Constants

- Interfaces can contain data fields
  - Data fields must be `public`, `static`, and `final`
- Interfaces contain constants
  - Provide a set of data that many classes can use without having to redeclare values

Creating Interfaces to Store Related Constants (cont'd.)

```
public interface PizzaConstants
{
   public static final int SMALL_DIAMETER = 12;
   public static final int LARGE_DIAMETER = 16;
   public static final double TAX_RATE = 0.07;
   public static final String COMPANY = "Antonio's Pizzeria";
}
```

Figure 11-29 The `PizzaConstants` interface

The Comparable Interface

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

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

- Then call to the method:
  - a.compareTo(b)

The Comparable Interface (cont)

- The compareTo method returns:
  - negative number *if b is greater than a*
  - zero *if a and b are the same*
  - positive number *if a is greater than b*.

- Implement the Comparable interface so that objects of your class can be compared, for example, in a sort method.

The Comparable Interface (cont)

- `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;
  }
. . .
}
```

The Comparable Interface (cont)

- 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.

The Comparable Interface (cont)

- Can you use the `Arrays.sort` method to sort an array of `String` objects?

- **Answer:** Yes, you can, because `String` implements the `Comparable` interface type.

Using The Comparable Interface

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

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

Using The Comparable Interface

- Now, we can write a call to the max method that computes the larger of two bank accounts (or any objects that implement the comparable interface), then prints its balance.

```
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 `object specific` method, like getBalance()

Creating and Using Packages

- Package
  - A named collection of classes
  - Easily imports related classes into new programs
  - Encourages other programmers to reuse software
  - Helps avoid naming conflicts or collisions
  - Gives every package a unique name

Creating and Using Packages (cont'd.)

- Compile the file to place in a package
  - We can use the export option, however, the parameters need to be changed
    - We are going to demonstrate them in the demo

- Package-naming convention
  - Use your Internet domain name in reverse order

Creating and Using Packages (cont'd.)

- **Java ARchive (JAR) file**
  - A package or class library is delivered to users as a JAR file
  - Compresses and stores data
    - Reduces the size of archived class files
  - Based on the Zip file format

- Demo

Creating and Using Packages (cont'd.)

- After the compiled jar file has been created, we can use it in a separate project

- To do so, we have to include the jar file in the classpath of the project
  - All the public classes will be available in the current project
  - Make sure not to delete the jar file, otherwise, the project won't be able to use it

- What happens to the protected instance variables?

Thank you

- Please let me know if you have any questions