Inheritance

Lecture 13
CGS 3416 Spring 2017

March 27, 2017

Subclasses and Superclasses

- **Inheritance** is a technique that allows one class to be derived from another.
- A derived class inherits all of the data and methods from the original class.

Example: Suppose that class Y is *inherited* from class X.

- class X is the **superclass**. Also known as *base class* or *parent class*.
- class Y is the **subclass**. Also known as the *derived class*, or *child class*, or *extended class*.
- class Y consists of anything created in class Y, *as well* as everything from class X, which it inherits

Declaring a subclass

Use the keyword `extends` to declare the derived class.

Example 1

```
public class AAA      // base class
{ ...  }

public class BBB extends AAA      // derived class
{ ...  }
```

Example 2

```
public class Employee {...}      // base class

public class HourlyEmployee extends Employee { ...  } //
derived class
```

The keyword super

- When you create a derived object, the derived class constructor needs to invoke the base class constructor.
- Do this with the keyword **super** – in this context, it acts as the call to the base class constructor.

```
super(); // base class default constructor
super(parameters); //base class parametrized
                   constructor
```

- The call to `super()` must be the first line of the derived class constructor.
- If explicit call to parent constructor not made, the subclass' constructor will *automatically* invoke `super()`. (the default constructor of the base class, if there is one).
- Can also use super to invoke a method from the parent class (from inside the derived class). Format:

```
super.method(parameters)
```

Example

```
//class HourlyEmployee, derived from Employee
public class HourlyEmployee extends Employee
{
     public HourlyEmployee() // default constructor
     {
          super(); // invokes Employee() constructor
     }

     public HourlyEmployee(double h, double r)
     {
          super(h,r); // invokes Employee constructor
                      w/ 2 parameters
     }

          // ...  more methods and data
```

The protected modifier

- Recall that **public** data and methods can be accessed by anyone, and **private** data and methods can be accessed only by the class they are in.
- **protected** data and methods of a public class can be accessed by any classes derived from the given class (this is also true in C++).
- In Java, a protected member can also be accessed by any class in the same `package` (to be discussed later)

The final modifier

In addition to creating constant variable identifiers, the keyword final can be used for a couple of special purposes involving inheritance:

- When used on a class declaration, it means that the class cannot be extended. (i.e. it cannot become a parent class to a new subclass).
- When used on a method declaration, it means that the method cannot be overridden in a subclass. (i.e. this is the final version of the method).

Method Overriding

Although the derived class inherits all the methods from the base class, it is still possible to create a method in the derived class with the same signature as one in the base. Example:

- Suppose a class Bird is derived from class Animal.
- Animal has a method:

```
void Sleep() { ... }
```

- We can define a method in class Bird with the same signature. The derived class version will *override* the base class version, when called through an object of type Bird.

```
Bird b = new Bird(); // create a
          Bird object which has all the
          Animal methods available.
b.Sleep(); // invokes the Sleep method from the
          Bird class
```

Method Overriding

Note that the Bird class' `Sleep()` method can still invoke the superclass' method, with the keyword `super`

```
public void Sleep()
{
     super.Sleep(); // invoke parent's Sleep()

     // continue with any processing specific
        to Bird
}
```

Casting

When a class B extends a class A, then an instance of the B class is of type B, but also of type A. Thus, such an instance can be used in all cases where a class B or class A object is required.

However, the reverse is not true! An instance of the class A is of course of type A, but it is not of type B.

Thus, we can use casting between the instances of classes. The cast inserts a runtime check, in order for the compiler to safely assume that the cast is used properly and is correct. If not, a runtime exception will be thrown.

```
Animal a2 = new Bird(); // create a
          Bird object which has all the
          Animal methods available.
a2.sleep(); //// invokes the Sleep method from the
          Bird class
```

Abstract Classes

- Superclasses are more general and subclasses are more specific.
- Sometimes a base class is so general that it doesn't make sense to actually instantiate it (i.e. create an object from it).
  - Such a class is primarily a grouping place for common data and behaviors of subclasses – **an abstract class.**
- To make a class abstract, use the keyword `abstract` (which is a modifier)

```
public abstract class Animal
```

- Now that Animal is abstract, this would be illegal:

```
Animal s = new Animal();
```

- Specifically, it's `new Animal();` that is illegal.

Methods can be abstract as well

- An abstract method is a method signature without a definition.
- Abstract methods can only be created inside abstract classes.
- The main purpose of an abstract method is to be overridden in derived classes (with the same signature)
- Example:

```
public abstract class Animal
      // Animal is an abstract class
{
      public abstract double eat();
         // eat is an abstract method

      // other methods and data
}
```

The Object class

In Java, **every** class is derived automatically from a class called `Object`. If no specific inheritance is declared for a class, it automatically has Object as a superclass.

While there are several methods in class `Object`, here are three important such methods, inherited by every Java class.

- `public boolean equals(Object object)`
- `public String toString()`
- `public Object clone()`

Let's look at each.

public boolean equals(Object object)

Tests whether two objects are equal. Returns `true` if equal, `false` if not. object1 and object2 same class type.

```
object1.equals(object2)
```

Default implementation is:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

Note that this default implementation is equivalent to the == operator, since it only tests the reference variables for equality. The intent is that subclasses of Object should override the `equals` method whenever they want a test of equality of two objects' **contents**.

public String toString()

Returns a string that represents the object. Call format:

```
objectName.toString();
```

The default version of the string might not always be useful, but this can be overridden in any derived class. Example for a class called Fraction:

```
public String toString()
{
    return numerator + "/" + denominator;
}
```

public String toString()

Assuming the above function for a Fraction class, the following illustrates its usage:

```
Fraction f1 = new Fraction(4,5);
          // create the fraction 4/5
System.out.print(f1.toString());
          // will print "4/5"

System.out.print(f1);
          // also prints "4/5" as this always invokes
          a class' toString method
```

public Object clone()

Remember, direct assignment between object names will only copy one reference variable to another. Use the `clone()` method to make copies of objects.

```
newObject = someObject.clone();
```

Not all objects can be cloned. Only objects imeplementing the `java.lang.Cloneable` interface (which will be discussed later) can use the clone method.

The `clone()` method from the object class does a "shallow copy" (i.e. copies reference variables verbatim). If a "deep copy" is needed (a la copy constructors in C++), you should override `clone()` for a class.

Other methods from class Object

- `finalize` – called by garbage collector to perform to perform cleanup on an object. Can be overridden, but rarely done.
- `getClass` – returns an object of type Class, with information about the calling object's type.
- `hashCode` – returns hash value that can be used as a key for the object (for use in a hash table, for example).
- `notify, notifyAll, wait` – related to multithreading.