Classes and Objects

Lecture 12
CGS 3416 Spring 2016

March 3, 2016

Classes and Objects

- An **object** is an encapsulation of data along with functions that act upon that data.
- It attempts to mirror the real world, where objects have attributes and are associated with activities.
- An object consists of:
  - **Name:** the variable name we give it.
  - **Attributes (or state)** a set of data fields and their current values, describing the object.
  - **Methods (behavior)** a set of methods that define the behavior aspects of the object.
- A **class** is a blueprint for objects. A class is a user-defined type that describes and defines objects of the same type.
- A class contains a collection of data and method definitions.
- An object is a single **instance** of a class. You can create many objects from the same class type.

Creating Objects

Objects are created from a class by using the new operator, and they must be attached to a reference variable. Two steps:

1. Declare the object reference variable.
2. Create the object with the new operator and attach it to the reference variable.

Format

```
ClassName objectReference;
objectReference = new ClassName();
```

(or combined into one statement)

```
ClassName objectReference = new ClassName();
```

Examples:

```
Circle myCircle;      // where Circle is a class
myCircle = new Circle();
Dog fido = new Dog();      // where Dog is a class
```

Creating Objects

Since the name used to refer to an object is a *reference variable*, and not the object itself, it is important to note that any assignments done on such a variable are just on the reference.

For example, if we create two objects, and then assign their variables together:

```
Circle c1 = new Circle();
Circle c2 = new Circle();

c1 = c2;
```

The last statement (c1 = c2) does not copy circle c2 into c1.

Instead, it copies the reference variable c2 to c1, which means that both reference variables are now referring to the same object (the second one, c2).

Using Objects

Once an object is created, access the object's internal methods and data with the dot-operator. Format:

```
objectReference.data
objectReference.method(arguments) //method call
```

Example:

```
Circle c1 = new Circle();
c1.radius = 10; // access radius instance variable

//compute and print the area with the findArea method
System.out.print("Area = " + c1.findArea());
```

Protection levels in a class (Visibility Modifiers)

We can declare members of a class to be **public** or **private**.

- public - can be accessed from inside or outside of the object.
- private - can only be used by the object itself.

The public members of a class make up the interface for an object (i.e. what the outside builder of the object can use).

The user of an object is some other portion of code (other classes, functions, main program).

Data Hiding

Although there is no set rule on what is made public and what is made private, the standard practice is to protect the data of a class by making it private.

Provide access to the data through public methods, which can maintain control over the state of the object.

Reasons for data hiding:

- Makes interface simpler for user.
- Principle of least privilege (need-to-know).
- More secure. Less chance for misuse (accidental or malicious).
- Class implementation easy to change without affecting other modules that use it.

Constructors

A constructor is a special member function of a class whose purpose is usually to initialize the members of an object.
A constructor is easy to recognize because:

- It has the same name as the class.
- It has no return type

Constructors can have parameters. A constructor without any parameters is known as a **default constructor**.
A constructor is automatically invoked when an object is created with **new**.

```
c1 = new Circle();// invokes default constructor

c2 = new Circle(9.0) //invokes constructor with one parameter
```

The usual purpose of a constructor is to perform any initializations on the object when it is created (primarily the instance variables).

Accessors and Mutators

Since it's a good idea to keep internal data of an object private, we often need methods in the class interface to allow the user of objects to modify or access the internally stored data, in a controlled way.

An accessor method is a function that returns a copy of an internal variable or computed value. A common practice is to name these with the word **get**.

A mutator method is a function that modifies the value of an internal data variable in some way. The simplest form of mutator function is one that sets a variable directly to a new value – a common practice is to name these with the word **set**.

Objects as Method Parameters

- Remember, objects are created with the new operator. The name we use is a reference variable.
- When an object is passed into a method, the reference variable is copied into the method's local parameter (just like with arrays) – method parameter becomes a reference to the original object.
- Bottom line: When an object is passed into a method (by its reference variable), the method has access to the original object. Changes to the object (from inside the method) will affect the original.

Class Variables and Methods

The modifier **static** can be used on variables and on methods.

- Variables
  - A static variable is shared by all instances of a class. Only one variable created for the class.
  - Instance variable (not static) – each object (i.e. each instance of a class) gets its own copy of such a variable.
- Methods
  - A regular method (instance method) can only be called by an object (an instance of the class).
  - A static method (class method) can be called without creating instances of a class. Called through class name or object name – but a better practice to call through the class name (to help remind that they are static). Example: `Math.round(x)`

Class Variables and Methods

The `static` and `final` qualifiers.

Access:

- Static variables can be accessed from both instance methods or static methods.
- Instance variables can not be accessed from static methods (since instance variables only exist when an object exists). Instance variables can be accessed from instance methods

To make a class variable constant, add the keyword `final` as a modifier on the declaration.

It's better to make your constants also `static` – since the value won't change, it's more efficient to have one variable shared by the entire class.

Example

```
class Student
{
     private int testGrade;
          // instance variable (non-static)
     private static int numStudents = 0;
          // static variable (class variable)
     private final static int pointsPossible = 100;
          // class constant

     public Student()
     {       testGrade = 0;      }

     public void setGrade(int gr)
     {       testGrade = gr;      }
```

Example (continued)

```
    public int getGrade()
    {       return testGrade;       }

    public static void incrementNumStudents()
    {       numStudents++;          }

    public static int getNumStudents()
    {       return numStudents;     }
}
```

Explanation

In this sample code:

- `testGrade` is an instance variable. Each object of type Student will have its own copy of `testGrade`
- `numStudents` is a class variable (static). There is only one variable shared by the whole class. The variables value can be changed, but changes are seen by all objects.
- `pointsPossible` is a class constant. There is only one variable (because of static), and its value cannot be changed)
- `setGrade` and `getGrade` are instance methods. They must be called through individual objects.
- `incrementNumStudents` and `getNumStudents` are static methods. They cannot access instance variables of the class, but they can be called through the class name, regardless of whether any objects have been created.

The Keyword this

- In Java, the keyword `this` is a reference variable to the current calling object (from inside an instance method). Once inside the instance method, this acts as the reference name for the calling object that you are in.
- In Java, this can also be used to call one constructor from another in a class, for the current calling object. Use it like the function name in the call. Example:

```
public Date(int m, int d, int y)//3 param constructor
{
     month = m;      day = d;      year = y;
}
public Date(int m, int d)// constructor with 2 params
{
     this(m, d, 0);// calls constructor with 3 params
}
```

Arrays of Objects

Creating an array of objects is a little trickier than an array of a primitive type.

1. Create an array using similar syntax to primitive types, but use the class name instead of the primitive type name:

   ```
   Student[] list = new Student[10];
   ```

   This only creates an array of reference variables – references for type `Student`.

2. Create the individual objects with new, and attach to the reference variables (the array positions). This can be done separately:

   ```
   list[0] = new Student();
   list[1] = new Student();
   ```

Arrays of Objects

It's easier to create the individual objects with a loop (as long as you are using the same constructor for each object):

```
for (int i = 0; i < list.length; i++)
     list[i] = new Student();
```

Each `list[i]` is the reference to an object now.