L4,5: Java Overview II

1. METHODS

Methods are defined within classes. Every method has an associated class; in other words, methods are defined only within classes, not standalone.

Methods are usually `public`. That is, any other class can call the method.

```
public <return-type> <methodname> (<paramater list>) {
...
}
```

Methods that are "internal" to the class and are not to be called from outside should be declared `private`.

```
private <return-type> <methodname> (<paramater list>) {
...
}
```

Methods can be called only on instances of a class (unless declared static as below). That is, you need to first create an instance of class in order to call a method on it.

```
Bicycle b = new Bicycle();
b.changeGear(2);
```

Within the class, a method is invoked by calling its name `<methodname>(<arguments>).`. Between classes you must use the dot operator ".": `<object>.<methodname>(<arguments>)`.

1.1 Constructors, multiple constructors

A constructor is a method that is invoked to create an object, like this:

```
SomeClass x = new SomeClass();
```

The constructor has no return type. It is implicitly `void`.

A class may have no constructor. In that case, Java provides a default "dummy" constructor (i.e. does nothing).

A class may have multiple constructors. When the user creates an object, the right constructor will be invoked that matches the types of the arguments. However, there cannot be two constructors with the same *signature* (list of parameter types).

```
Class Person {
   private String firstName, lastName;
   private int id;
   private String address;

   public Person(String fname, String lname)
   public Person(int pid)
   public Person(String address)
}

Person a, b, c;
a = new Person("Howard", "Zinn");
b = new Person(47561845);
c = new Person("8650 College Street Brunswick 04011 USA");
```

1.2 `main` method

Some classes are meant to be utilized by other classes. Others are meant to be stand-alone programs. A class that is a standalone program, i.e. the user can "run" it, it must have a `main` method:

```
public class AClass {
   public static void main(String args[]) {
       //this is the method that gets executed when the user runs the class
       //the body of the method
       ...
   }
}
```

Suppose we first compile this class:

```
javac AClass.java
```

Then we can run it:

```
java AClass
```

In Java, when you execute a class, the system starts by calling the class `main` method. Note that main is static, that is it does not need an instance of the class.

1.3 `equals` vs. ==

You have seen == on integers, etc. You can also use == on objects to determine whether two objects *refer to the same object.* That is, == compares *reference*, not values.

```
SomeClass x, y;

x = new SomeClass();
y = x;
//then x=== y  will return true
```

Any class has a(default) `equals()` method that is inherited from class `Object` (all classes inherit from it in Java). The default version checks for equality of reference.

Usually you are not interested whether the two objects refer to the same object, but rather, whether two objects are equal—i.e. their fields are equal. If you want to check data equality, you need to implement (override) it.

The `equals` method on class `String` checks for data equality. For example,

```
String s = "Bowdoin";
if (s.equals("bowdoin"))
```

will return `true`.

1.4 toString()

This is a method that returns the string representation of the object its called on. It is defined in class `Object` and therefore inherited by all objects.

For example,

```
class Point {
     private int x, y;

     public Point(int x, int y) {
        this.x = x;
        this.y = y;
     }
     public int getX() {
        return x;
     }
     public int getY() {
        return y;
     }
}
public class ToStringDemo {
     public static void main(String args[]) {
     Point p = new Point(10, 10);

     // using the Default Object.toString() Method
     System.out.println("Object toString() method : "+p);

     // implicitly call toString()
     String s = p + " testing";
     System.out.println(s);
     }
}
```

Try playing with this.

If you want more useful information about an object (for e.g. helpful in debugging) you'll have to override the `toString` method:

```
public void toString() {
  System.out.println("x=" + x + ", y=" + y);
}
```

2. STATIC VARIABLES AND METHODS

When a number of objects are created from the same class, they each have their own distinct copies of instance variables, stored in different memory locations.

A class can have variables that are declared `static`. These variables are associated with the class, not with the instances of the class.

The static variables are common to the class and are shared by all instances of the class. Any object can change the value of a class variable; class variables can also be manipulated without creating an instance of the class.

A static variable is accessed as `<class-name>.<variable-name>`.

Example: a BankAccount class can keep a static variable for the number of accounts open. Every call to a constructor would increment this variable.

A common use of static variables is to define "constants". Examples from the Java library are `Math.PI` or `Color.RED`.

A method can also be declared `static`. Then it is associated with the class, and not with an object. From inside a class it is called by its name, just like a non-static method. From ouotside the class it is called on the class `<class-name>.<method-name>(<arguments>)`.

You may ask: when to declare a method static? When it performs a function specific to the class, not to a specific object.

```
public class SomeClass {
    ...
   //returns true if n is prime
   public static boolean isPrime(int n) {
       ...
   }
}
```

To find whether a number is prime, you dont need to create a specific object. From outside the class, call `SomeClass.isPrime(645)`.

A common use for static methods is to access static variables.

The functions is `Math` are all static: `Math.sin(x), Math.cos(x), Math.exp(d), math.pow(d1, d2)`.

Note: static methods cannot acces instance variables (there is no instance), and cannot access `this` (there is no instance for this to refer to).

3. SCOPE

Types of variables in Java:

(1) instance variables (non-static)
(2) class variables (static variables)
(3) local variables
(4) parameters

Scope: Within a method. Between methods. Between classes.

Variables must be declared within a class (instance variables or global variables) or a method (local variables).

An instance variable is visible/accessible to all methods of that class. If it is public, it is also visible outside the class.

A local variable is visible only inside the method; in the code following its declaration, within the tightest enclosing .

The parameters of a method are visible (only) inside the method and do not need to be re-declared inside.

3.1 `this`

Used to refer to an instance variable that clashes with a parameter:

```
class A {
   private int age;

public void setAge(int age) {
   this.age = age;
}
```

3.2 Example

```
public class ScopeTester {

    //instance variable
    private int x = 1;

    public void test1 (int x) {
//a parameter shadows an instance variable
System.out.println("x = " + x);
x = 10;
System.out.println("x = " + x);
System.out.println("this.x = " + this.x);
    }

    public void test2() {
//you can redeclare an instance variable
double x = 10.3;
System.out.println("x=" + x);
System.out.println("this.x=" + this.x);
    }

    public void print() {
System.out.println("x=" + x);
    }

    public boolean equals (ScopeTester s) {
return (x == s.x);
    }

    public static void main (String args[]) {
ScopeTester s = new ScopeTester();

int a = 100;
System.out.println("before test1: a = " + a);
s.test1(a);
//a is changed in test1(), but not visible outside
System.out.println("after test1: a = " + a);

s.test2();
s.print();
ScopeTester x, y;
x = new ScopeTester();
y = new ScopeTester();
if (x==y) System.out.println("same object");
else System.out.println("NOT same object");
if (x.equals(y)) System.out.println("equal");
else System.out.println("not equal");

System.out.println("this is the default toString(): " + x);


    }
}
```

4. INHERITANCE

Object-oriented programming allows to define class hierarchies. That is, it allows for classes to inherit from other classes, which inherit from other classes, and so on.

```
class A {
   //variables
   //methods
}
class B extends A {
  //class B inherits ALL variables and methods of class A
}
```

We say that class A is the parent of class B, or the *super-class* of class B; class B is the *sub-class*. A class can inherit from a single other class.

Example:

```
class OneDPoint

class TwoDPoint extends OneDPoint

class ThreeDPoint extends TwoDPoint
```

The sub-class inherits all the variables and methods in the super-class. It is as though the variables and methods of the super-class would be textually present in the sub-class.

For example:

```
class Person {
   String name;  //the name

   void Person(String s) {
       name = new String(s);
   }
   String getName() {
      return name;
   }
};
class Student extends Person {

  int class; //the class of graduation

  void Student(String n, int year) {
    super(n);
    class = year;
  }
}
```

It is valid:

```
class Student s = new Student("Jon Myers);
System.out.println("name is " + s.getName());
```

In the constructor of the sub-class, the first thing to do is invoke the constructor of the super-class.

In our toy example above this makes no sense because class Person is empty and it does not even have a constructor. But imagine that we had to do some work.

—Discuss the hierarchy in Java: class `Object`. Everything inherits from it.

—Discuss overriding methods.

5. DYNAMIC TYPE-CASTING

Java performs automatic type conversion from a sub-type to a super-type. That is, if a method requires a parameter of type A, we can call the method with a parameter that is a sub-class of A. Basically this says that any class that inherits from A can be used in place of type A.

6. PROGRAMMING WITH ASSERTIONS

A good programming practice in any language is to use assertions in the code to enforce invariants.

For example, if at some point in a method we know that a variable say x must be non-zero, we write:

```
 assert (x != 0);
```

When the code is executed, if x is non-zero, the assertion succeeds. Otherwise, if x is zero, the assertions fails, and the program terminates with an error.

In order to enable assertion you have to compile with the flag `-ea`:

```
javac -ea xxx.java
```

Assertions don't have any effect on the code as such, they are just used to make sure that certain conditions are true. It is good pratice and style to include assertions in your code. They help you catch errors earlier.

7. JAVA INTERFACES

So far we talked about the interface of a class, and we said that we first figure out the interface, then we go and put in the details.

Java lets you take this one step further; it lets you declare an `interface`, which is like a class, just that it does not give the implementation, just the *signatures* of the methods. All methods in an interface must be public.

```
public interface xxx {

// list the signatures of the methods
}
```

Interfaces are used to describe a class, but without commiting to an implementation. An interface is usually followed by at least one class that *implements* the interface. Every class that implements an interface must define all methods in the interface.

For example, here is the set of methods that we'd expect from a `List`, irrespective of the implementation:

```
public interface List {

  //return the size of the list
  public int size();

  //return true if list is empty
  public boolean isEmpty();

  //return true if the list contains the given element
  public boolean contains(Object value);

  //add one element to the list
  public void add(Object value);

  //remove the element from the list
  public void remove(Object value);
}
```

If we were to write an implementation of `List`:

```
public class ArrayList implements List {

  //... implements all methods in the interface, and possibly more
}
```

Note: Why use `Object` class? Note that we do NOT know the type of the objects in the list!! class `Objects` is a superclass for all classes in Java.

When compiling a class that implements an interface, the compiler checks to see that *all* methods specified in the interface are implemented; the parameter types and return type must match; if not, it gives an error. So you can think of an interface is a way of specifying a set of methods that an object must support. An interface is a contract between a class and the outside world.

It is possible that a class implements several interfaces—then it needs to implement all methods in all interfaces.

You probably won't be defining interfaces, but you'll see classes that implement interfaces.

8. TESTING AND DEBUGGING

If your program compiles, that does not mean it is correct. It can still have bugs... that is, logical errors. You will find that you will spend most of your time debugging your code. There is no recipe on how to debug — just plenty of print statements, and/or use BlueJ.

To find all possible bugs, you need to *test* your program on different inputs, and take into account all situations that may appear.

Graphics in Java

Java has two libraries for creating GUIs (graphical users interfaces): `awt` and `swing`. The Swing toolkit is newer, and richer. We'll be using both.

```
import javax.swing.*
import java.awt.*
```

A GUI application consists of individual components you can interact with: buttons and menus and labels, text fields, drawing areas, icons. All these are called components.

To appear onscreen, every GUI component must be part of a containment hierarchy. A containment hierarchy is a tree of components that has a top-level container as its root.

Swing provides three container classes `JFrame, JApplet, JDialog` (defined in `javax.swing.*`) which allow the programmer to create and handle windows.

Today we'll learn about a `JFrame`. A `JFrame` is an object that can handle a window on screen, which has a toolbar and a border, can handle mouse events, colors, buttons, etc. It has a canvas on which it can draw things.

A class that needs to do graphics will inherit from `JFrame` (or one of the other containers listed above).

To pick up mouse events a class has to *implement* `MouseInputListener` (which extends `MouseListener` and `MouseMotionListener`). This is an *interface* —- a list of methods that must be implemented. When a class promises to implement an interface this basically means that it makes a contract to implement the set of methods specified in the interface.

In this case, the mouse handling methods specified in `MouseInputListener` are the following.

```
import java.swing.*
import java.swing.event.*
import java.awt.*
import java.awt.event.*

public class xxx extends JFrame implements MouseInputListener {

   ...

   public void mousePressed(MouseEvent e);

   public void mouseDragged(MouseEvent e);

   public void mouseReleased(MouseEvent e);

   public void mouseClicked(MouseEvent e);

   public void mouseEntered(MouseEvent e);

   public void mouseExited(MouseEvent e);
   public void mouseMoved(MouseEvent e);
}
```

Note that in order to handle mouse events you need to import `java.swing.event.*` and `java.awt.event.*`.

Your program will control: the window size, what gets drawn, when, what happens when the mouse is clicked, released, moved, dragged, etc. The constructor will usually contain something like this:

```
super("name of window");
setSize(400, 300);

//exit the program when the window is closed
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

...
//whatever else needed in the constructor

//if handling mouse then
addMouseListener(this);
addMouseMotionListener(this)

setVisible(true);
```

In order to draw in the window, one needs to grab the canvas of the `JFrame`, which is of type `Graphics`:

```
Graphics g = this.getGraphics();
```

Note: `this` refers to the current object.

Check the documentation for the methods supported by `Graphics`. Here are some of them. They must be called on a Graphics object.

—drawLine(Point p1, Point p2)

—drawImage(...)

—drawOval(), drawPolygon(), drawRect, filArc(), fillOval, etc

—getColor(), setColor(), getFont(), setFont() etc

The Java coordinate (0,0) is in the upper left corner.

To put an object on a canvas:

```
Graphics g = this.getGraphics();
...
g.drawLine(..)
```

9. IN-CLASS PROGRAMMING

An application that lets you scribble.

Reading

—Bailey Appendix B

—Bailey chapter 1, 2

—BlueJ tutorial
[If you need more background: Eventful: chapter 1-8, chapter 9, 13, 14, 16, 17]