COE318 Lecture Notes Week 3 (Week of Sept 15, 2014)

Updated:

Added links to Robot class.

Added test class for Robot.

Announcements (REPEAT!)

- Quiz (5% of total mark) on Wednesday, September 24, 2014. Covers weeks 1–3 and labs 1–3.
- Midterm (20% of total mark) on Wednesday, October 15. Covers weeks 1–6.
- Note: Week 2 lecture notes have been updated. (Material not covered last week has been moved to this week's notes.)
- Quizzes (with answers) for the previous two years are available on the webpage

Topics

- Example: Robot (a *mutable* class)
- Example: Person (an *immutable* class)
- Notes on Lab 3
- Java nitty-gritty details
- A closer look at variables
- Text/Tutorial sections covered
- Q&A

Example: The `Robot` class

- We wish to simulate robots that can move around on a flat surface.
- Each robot should know where it is. So we need instance variables for its position (we use its x and y coordinates.)
- Each robot should also respond to commands such as "goNorth" or "goWest".
- Robots should also be able to calculate their distance from a specific grid point or from some other robot.

```
package coe318.week2;


public class Robot {
  //Instance variables track the Robot's position

  private int x;
  private int y;

  /**
   * Create a Robot with the specified initial position
   *
   * @param xInit
   * @param yInit
   */
  public Robot(int xInit, int yInit) {
    x = xInit;
    y = yInit;
  }

  /**
   * Get the Robot's current x position.
   *
   * @return x position
   */
  public int getX() {
    return x;
  }

  /**
   * Get the Robot's current y position.
   *
   * @return y position
   */
  public int getY() {
    return y;
  }

  /**
   * Make Robot take a step North.
   */
  public void goNorth() {
    y++;
  }

  /**
   * Make Robot take a step South.
   */
  public void goSouth() {
    y--;
  }

  /**
   * Make the Robot take one step East.
   */
```

```
  public void goEast() {
    x++;
  }

  /**
   * Make the Robot take a step West.
   */
  public void goWest() {
    x--;
  }

  /**
   * Get the distance from the given point to the Robot.
   *
   * @return the distance
   */
  public double getDistanceFrom(int xx, int yy) {
    int x2 = x - xx;
    x2 = x2 * x2;
    int y2 = y - yy;
    y2 = y2 * y2;
    return Math.sqrt(x2 + y2);
  }

  public double getDistanceFrom() {
    return getDistanceFrom(0, 0);
  }

  public double getDistanceFrom(Robot other) {
    return getDistanceFrom(other.getX(), other.getY());
  }

  public static void main(String[] args) {
    Robot r2d2, c3po;
    r2d2 = new Robot(0, 0);
    c3po = new Robot(1, 1);
    System.out.println("r2d2 distance from origin: "
            + r2d2.getDistanceFrom());
    System.out.println("r2d2 distance from c3po: "
            + r2d2.getDistanceFrom(c3po));
    r2d2.goNorth();
    c3po.goEast();
    System.out.println("r2d2 distance from c3po: "
            + r2d2.getDistanceFrom(c3po));
  }
}
```

- The following class tests Robot:

```
package robotstaticarrays;
/** Demonstration using Robot objects.
```

```
 * @author kclowes
 */
public class RobotStaticArrays {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Robot r2d2, c3po;
        Robot [] robots;
        r2d2 = new Robot(0, 0);
        c3po = new Robot(1, 1);
        robots = new Robot[2];
        robots[0] = r2d2;
        robots[1] = c3po;

        r2d2.goNorth();
        robots[1].goEast();
        System.out.println("r2d2 distance from c3po: "
                + r2d2.getDistanceFrom(c3po));
        System.out.println(Robot.getNumRobotsCreated() +
                " created");
    }
}
```

- You can download the source code files Robot.java and RobotStaticArrays.java.
- You can also download the zipped Netbeans project RobotStaticArrays.zip

Overloading method names

- Note that there are 3 different versions of the method `getDistanceFrom()`:
  1. One that gets the distance from a specified point.
  2. Another that gets the distance of the robot from the origin.
  3. Yet another that gets the distance from the robot to another robot.
- This is called ***overloading*** the method name.
- Two or more methods can have the same name as long as the number or types of parameters are distinct.

- Note that this is completely illegal in C where you cannot have two functions with the same name even if they have different parameters.

Other remarks

- The instance variables (x and y) are initialized when a Robot object is created (with its c*onstructor*).
- The instance variables determine the *state* of the object. Since the instance variables can change (with methods like `goSouth()`), we say that the object is *mutable*.
- It is common for instance variables to have names that are *nouns*.
- Because the instance variables are declared as *private*, other objects cannot directly access the variables or change them.
- Other objects can, however, use methods (like `getX()`) to get a copy of an instance variable.
- But there is not method like `setX(int x)` that would allow another object to directly movve the object.
- Other objects can only move a robot one step at a time in one of 4 directions.
- Note also that the names of methods are usually best given as *verbs* (because the are commands given to the object to behave in some way.)

Example: the Person class

- This simple class is used to define Person objects where each Person has a name and a gender.
- Once created, the Person's name and gender cannot be changed.
- This is an example of an *immutable* class.
- Immutable objects are preferable to mutable objects; they make programs more reliable by reducing possible sources of bugs.
- For example, when a mutable object is passed to another object, that object can modify the passed object's state. If the other object comes from an unreliable or malicious source, this can create havoc.
- This kind of bug cannot occur for immutable objects.
- It is not always obvious that an object can be made immutable. Sometimes the natural way to design the class is to make it mutable and the immutable version requires more work from the programmer. (One of the questions shows an example of this.)
- One design decision to note: a `boolean` data type is used to indicate gender using the instance

variable `isMale`. In general, if you have an instance variable that has only two possible values, consider using a boolean data type.

- By the way, it is a common convention to use a variable name "isWhatever" for boolean data types. (For example, if some kind of object modelled the weather, a boolean instance variable named `isRaining` could indicate whether an umbrella would be advisable.)

```
public class Person {
    private String name;
    private boolean isMale;

    /** The following is a CONSTRUCTOR.  It is
        invoked with parameters when a new Person
        object is created.
    */
    public Person(String n, boolean x) {
        name = n;
        isMale = x;
    }

    public String getName() {
        return name;
    }

    public boolean isMale() {
        return isMale;
    }

    public String toString() {
      return name + "(" +
                  (isMale ? "M" : "F") + ")";
    }


    public static void main(String[] args) {
        System.out.println(
                 new Person("Jane Smith", false));
        System.out.println(
               new Person("Alice Cooper", true));
    }
}
```

- The `toString()` method gives a String (human-readable) description of a specific Person

object.

- The line:
  ```
  name + "(" +  (isMale ? "M" : "F") + ")"
  ```
  uses the + operator to join (or *concatenate*) strings together and the ? operator is used to indicate M or F depending on the person's gender.

Java nitty-gritty details

- Netbeans (or other IDEs such as Eclipse) hide underlying details about Java.
- But you can write and run Java software without an IDE.
- To do so, however, you need to understand some basic rules about Java.
- First, Java is a *compiled* language. So Java source code has to be compiled (without any errors) before it can be run.
- The java source code file must contain exactly one *public class ClassName* declaration.
- The source code file **must** have the name *ClassName* (and this is case-sensitive) and must have the extension *java*.
- For example, if the source code file begins with:

```
public class Foo {
```

then the full name of the source code file must be `Foo.java`

- The source code file is then compiled with the command `javac Foo.java`
- If there are no errors, the compiled (executable) file is produced: `Foo.class`
- This file can then be executed with: `java Foo` (Note: you do not add the .class extenstion.)
- The class file, however, is not a binary executable (as would be produced by a C compiler) customized to the particular hardware and operating system you are using.
- Rather, the class file is executed by the *Java Virtual Machine* (JVM) which simulates the virtual machine defined by Java. In short, it *interprets* the class file.
- Although the Java Machine Language is designed so that it can be interpreted very efficiently, it cannot run as fast as a binary executable file meant for the actual hardware.
- To get around this limitation, the JVM maintains statistics about how the code is used and determines which sections of code consume the most time.
- The run-time system then dynamically translates these sections to the actual underlying machine language. This is done by the *Just In Time* (JIT) compiler.
- Once this optimization is done, Java programs can run as fast and sometimes faster than traditionally compiled programs written in C.
- Alas, there is a cost. One of the annoying features of Java is the slow start up time. (You may have noticed that Netbeans — which is written entirely in Java — takes a long time to get

started. This is because a lot of classes have to be loaded and the JIT needs time to do its work.)

- This is one reason why not many desktop applications are written in Java.
- However, *server-side* applications (using the *Enterprise Edition* of Java) do not have the same problem because:
  - They do basic computation and do not need to load all the large classes associated with a graphical user interface (GUI).
  - Furthermore, server-side applications run for days, months or even years. In these cases, the start-up time penalty is irrelevant.)
- The start-up time penalty is also greatly reduced for the lighter-weight JVM used on portable devices such as cell phones, blu-ray players, PVR's or tablets. (e.g. Nokia, Blackberry, Android, Sony)

Variables: a closer look

- There are different types of variables in Java:
  - **instance variables**: associated with Objects.
  - **passed parameters**: passed to methods as parameters.
  - **locals**: local and visible only with the execution of a method.
  - **static (**or **class) variables**: associated with a class itself and with any objects (if any) of the class's data type.
- We now examine these different ways of using variables, including where the variables are stored in memory and what their lifetime is.

What is common to ALL variables?

- Any variable must be *declared* before it is used.
- The *declaration* gives the variable a *type* and a *name*. And, since Java is a strongly-typed language, the type cannot be changed (although it may be "cast", a topic we will discuss later on.)
- The name of variable should be sensible. Choosing a good name makes it easier to write, read and maintain code.
- By convention, variable names should be lower case. If the sensible name is more than one word long, the *camel case* convention should be followed. For example, if a variable describes

a person's "hair colour", a sensible name would be `hairColour`.

- One exception to this convention is for constants; it is common practice to give the name in all uppercase. For example, in the source code Math.java we read:

```
public static final double PI = 3.141592653589793;
public static final double E = 2.718281828459045;
```

What data types are available?

- There are two different kinds of data types in Java:

1. **Primitive**: there are 8 primitive data types in Java (boolean and 7 numerical types):

   i. `byte` (8 bit signed integer)

   ii. `short` (16 bit signed integer)

   iii. `int` (32 bit signed integer)

   iv. `long` (64 bit signed integer)

   v. `float` (32 bit floating point number)

   vi. `double` (64 bit floating point number)

   vii. `char` (Unicode character or 16 bit unsigned integer)

   viii. `boolean` (`true` or `false`)

2. **Reference**: these data types refer to *objects*. The Java Language specifies two kinds of reference types: `Object` and `String`. However, every Java source code file that defines a class also defines a new reference data type: the class's name.

   - In addition to String and Object, the Java "standard library" defines thousands of other reference data types (classes).
   - And every time a programmer writes a new java file, an additional data type is defined.
   - In short, there is an *unlimited* number of reference data types.

- By convention, class names (hence reference data types) begin with an uppercase letter whereas all primitive data types are lowercase.
- Thus, if you see:

```
byte var1;
Byte var2;
```

- It is reasonable to assume that `var1` is a primitive data type and `var2` is a reference data type. And, yes, every primitive data type has a corresponding reference data type.

How is data stored in memory?

- The value of any variable must be somewhere in memory. But this is done quite differently for primitive and reference variables.
- The value of a primitive variable is stored in a block of memory equal to the size of the primitive data type. (For example, a byte variable needs 1 byte of memory whereas a long or double variable needs 8 bytes.)
- Reference variables, on the other hand, always take up the same amount of memory space: 4 bytes (or 8 bytes on 64-bit implementations of the JVM).
- To be clear, suppose we write `Robot r2d2 = new Robot(1, 2);`
  - There is a single variable `r2d2` of type `Robot`. So it is a reference data type.
  - Now the actual robot object will need more than 8 bytes of memory, yet the variable itself needs only 4! How does this work?
  - The Robot object itself (and **all** objects of any type) reside in a special area of memory called the **heap**.
  - The `r2d2` reference variable simply holds the address of the starting location of the heap memory allocated to the actual Robot object.
- Since *instance variables* are part of an object, they must reside inside the memory allocated for the object. Hence all instance variables are found somewhere on the heap.

The Stack

- Objects and their instance variables reside in the heap.
- Local variables and parameters, however, reside elsewhere: on a data structure that can grow and shrink dynamically called the *Stack*.
- But first, let's recall with the following example the difference between instance variables, parameters and local variables:

```
public class C {
    private int i; //instance variable

    public void f(int j) { //j is a parameter
         int k;  //k is a local variable
         k = 2*j;
         i = i + k;
    }

    public void g() {
         int k; //another local variable named k
            //
```

```
    }
}
```

- When a method is invoked, the parameters are first pushed onto the stack. When the method is entered, additional space on the stack is reserved for all local variables. The method's code is then executed.
- While the code is being executed, the only variables available to it are its own local variables, its passed parameters and all of the object's instance variables. i.e the method has no way of accessing or modifying any local variables in the method that invoked it.
- When the method finishes, it releases the space on the stack occupied by its local variables and parameters: they no longer exist.
- Many methods execute very quickly – nanoseconds or microseconds – so the lifetime of locals and parameters is short indeed.
- The lifetime of instance variables, on the other hand, is usually much longer. These variables continue to exist as long as the object they belong to exists. For example, in the first 3 labs all of the objects created have been immortal and continue to exist as long as the program is running.

The Heap and Garbage Collection

- The JVM maintains a *reference count* of how many reference variables point to it.
- If the reference count drops to zero, the object is eligible for *garbage collection* which reclaims the space used by the object so that the memory on the heap can be used for some other object.
- Garbage collection relieves the programmer from the responsibility of releasing dynamically allocated memory.

Variable initialization

- Local variables are not automatically initialized; using them before they are set causes a compilation error.
- Instance variables are automatically initialized to 0 (for numbers), the character '\0' for chars and `null` for references. `null` (a keyword in Java) can represent a reference to *any* type of object but any attempt use the reference will result in a runtime error (`NullPointerException`).

Arrays are objects

- Arrays are always objects (even if they are arrays of a primitive data type).
- They are declared as in the following examples:

```
int [] arr;  //arr: array of ints; size not yet known
Robot [] robots; //robots: an array of Robots
```

- First we initialize their size:

```
arr = new int[5]; //arr: an array of 5 ints
robots = new Robot[3]; //an array of 3 Robots
```

- Next we initialize the contents of each element:

```
arr[0] = 3;
arr[1] = 1;
arr[2] = 4;
//etc
robots[0] = new Robot(1, 2);
robots[1] = new Robot(2, 3);
robots[2] = new Robot(1, 1);
```

- We can combine these steps if we want with:

```
int [] arr = {3, 1, 4, 1, 5};

Robot [] robots = {
    new Robot(1, 2),
    new Robot(2, 3),
    new Robot(1, 1)
}
```

Notes on Lab 3

- Read carefully sentences like: *If there is a left neighbour, the count is the sum of the digit and the modulus times the count of the left neighbour.*
- In this lab, you will build a fundamental and commonly used way to keep related things together: a *linked list*.

Questions

1. Consider the following code fragment:

```
Robot robbie;
Robot jane;
robbie = new Robot(0, 0);
jane = new Robot(0, 0);
robbie.goNorth();
jane.goSouth();
robbie.goEast();
```

```
robbie.goEast();
jane.goWest();
```

What are the x and y positions of the jane and robbie robots after this code is executed?

2. Modify the `Person` class so that the name instance variable is replaced by first and last names. The `getName()` method should return the first and last names separated by a space.
3. Suppose the Person class had an instance variable `private char gender`. Show the kind of errors that could occur.
4. Suppose that we wanted to add the method `int getAge()` to the Person class. This would require an additional instance variable. Explain how Person objects could still be make immutable. (You don't have to actually implement this. Just explain in general.)
5. Suppose you had to implement a different kind of robot called PointingRobot. It can do the same thing as Robot but the commands are different. Instead of having 4 different commands to move (goNorth(), goEast(), etc.), there is only one command: `void step()` which takes one step in the current direction. Another command `void setDirection(int dir)` sets the direction. (We assume 0 is North, 1 is East, 2 is South and 3 is West.)
   Write a class `PointingRobot` to implement this.
6. Suppose you have existing software that uses the Robot class. The Robot class was obtained from a company and you pay a license fee to use. The license expires and you don't want to buy an extension. You also have a PointingRobot class. Write a new Robot class with the identical API to the expired version that uses the PointingRobot class to actually make the robot move.(*Note*: This is an example of a very useful technique called an *adaptor classs*.)

Answers

1. **Answer:** robbie: x = 2; y = 1 jane: x = -1; y = -1
2.

```
public class Person {
    String first;
    String last;
    boolean isMale;

    /** The following is a CONSTRUCTOR.  It is
        invoked with parameters when a new Person
        object is created.
```

```
    */
  public Person(String fst, String lst, boolean x) {
        first = fst;
        last = lst;
        isMale = x;
    }

    public String getName() {
        return first + " " + last;
    }

    public boolean isMale() {
        return isMale;
    }

    public String toString() {
      return getName() + "(" +
                   (isMale ? "M" : "F") + ")";
    }


    public static void main(String[] args) {
        System.out.println(
             new Person("Joe", "Smith", false));
        System.out.println(
             new Person("Alice", "Cooper", true));
    }
}
```

3. If gender were a `char` data type, there would be more than 65,000 possible genders! (Note: in Java `char`'s are 16-bits unlike C chars which are 8 bits). That's too many possibilities! You might choose to use 'm' and 'f' for male and female, but how would you handle a user who mis-typed their gender as 'q' or 'π'?
4. If `age` were an instance variable the Person class would no longer be immutable since a person's age changes every year. However, a person's date of birth (dob) does not change and one can calculate their age at any time. This would allow the Person to remain an immutable class. The price to pay would be that the programmer would have to write a `getAge()` method that is non-trivial.
5.

```
package notes;
```

```
/**
 * A robot that can step() or setDirection().
 *
 * @author kclowes
 */
public class PointingRobot {

  private int x;
  private int y;
  private int direction;
  //This way of defining symbolic constants
  //is better than nothing, but an experienced
  //Java programmer would use an enum instead
  public final static int NORTH = 0;
  public final static int EAST = 1;
  public final static int SOUTH = 2;
  public final static int WEST = 3;

  /**
   * Take one step in the current direction.
   */
  public void step() {
    switch (direction) {
      case NORTH:
        y++;
        break;
      case EAST:
        x++;
        break;
      case SOUTH:
        y--;
        break;
      case WEST:
        x--;
        break;
      default:
        System.exit(-1);
    }
  }

  public int getDirection() {
    return direction;
  }

  /**
   * Set the direction (must be 0, 1, 2, or 3)
```

```
   * @param direction
   */
  public void setDirection(int direction) {
    this.direction = direction % 4;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  @Override
  public String toString() {
    return x + ", " + y;
  }

  public static void main(String[] args) {
    PointingRobot r1, r2;
    r1 = new PointingRobot();
    r2 = new PointingRobot();
    r1.step();
    r1.step();
    r1.setDirection(PointingRobot.WEST);
    r1.step();
    r2.setDirection(SOUTH);
    r2.step();
    System.out.println("r1: " + r1);
    System.out.println("r2: " + r2);
  }
}
```

6.

```
package notes;

/**
 * Adaptor allowing original Robot to use a
 * PointingRobot
 *
 * @author kclowes
 */
public class Robot {
```

```
PointingRobot rob;

public Robot(int x, int y) {
  rob = new PointingRobot();
  while( x != rob.getX()) {
    if(x > rob.getX()) {
      rob.setDirection(PointingRobot.EAST);
    } else {
      rob.setDirection(PointingRobot.WEST);
    }
    rob.step();
  }
  while(y != rob.getY()) {
    if(y > rob.getY()) {
      rob.setDirection(PointingRobot.NORTH);
    } else {
      rob.setDirection(PointingRobot.SOUTH);
    }
    rob.step();
  }
}

public int getX() {
  return rob.getX();
}

public int getY() {
  return rob.getY();
}

public void goNorth() {
  rob.setDirection(PointingRobot.NORTH);
  rob.step();
}

 public void goSouth() {
  rob.setDirection(PointingRobot.SOUTH);
  rob.step();
}
 public void goEast() {
  rob.setDirection(PointingRobot.EAST);
  rob.step();
}
 public void goWest() {
  rob.setDirection(PointingRobot.WEST);
```

```
    rob.step();
  }


  @Override
  public String toString() {
    return rob.toString();
  }

  public static void main(String[] args) {
    Robot r = new Robot(2, 4);
    r.goWest();
    System.out.println("r: " + r);
  }
}
```