CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 133 – Object-Oriented Computer Graphics Programming
Spring 2015
Clevenger/Muyan

Assignment #1: Class Associations

Due Date: Thursday, February 26th [3 weeks]

Introduction

This semester we will study object-oriented graphics programming by developing a program which is an animated car-racing game. In this game you will be controlling a race car driving around a track, trying to avoid collisions with other cars while keeping your car fueled and maneuvering around obstacles such as oil slicks.

The goal of this first assignment is to develop a good initial class hierarchy and control structure by designing the program in UML and then implementing it in Java. This version will use keyboard input commands to control and display the contents of a "game world" containing the set of objects in the game. In future assignments many of the keyboard commands will be replaced by interactive GUI operations, and we will add graphics, animation, and sound. For now we will simply simulate the game in "text mode" with user input coming from the keyboard and "output" being lines of text on the screen.

Program Structure

Because you will be working on the same project all semester, it is extremely important to organize it correctly from the beginning. Pay careful attention to the class structure described below and make sure your program follows this structure accurately.

The primary class in the program encapsulates the notion of a ***Game***. A game in turn contains several components, including (1) a ***GameWorld*** which holds a collection of *game objects* and other *state variables*, and (2) a set of methods to accept and execute user commands. Later, we will learn that a component such as *GameWorld* that holds the program's data is often called a *model*.

The top-level *Game* class also manages the *flow of control* in the game (such a class is therefore sometimes called a *controller*). The controller enforces rules such as what actions a player may take and what happens as a result. This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

In this first version of the program the top-level **Game** class will also be responsible for displaying information about the state of the game. In future assignments we will learn about a separate kind of component called a *View* which will assume that responsibility.

The program also has a class named ***Starter*** which has the **`main(String[]args)`** method. **`main()`** does one thing – construct an instance of the ***Game*** class. The game constructor then instantiates a **GameWorld**, calls a **GameWorld** method **`initLayout()`** to set the starting layout of the game, and then starts the game by calling a method **`play()`**. The **`play()`** method then accepts keyboard commands from the player and invokes appropriate methods to manipulate the game world and display the game state.

The following shows the pseudo-code implied by the above description. It is important for things that we will do later that your program follows this organization:

```
class Starter {
   main() {
      Game g = new Game();
   }
}
```

```
class GameWorld {
   public void initLayout(){
      //code here to create the
      //initial game objects/layout
   }
   // additional methods here to
   // manipulate world objects and
   // related game state data
}
```

```
class Game {
   private GameWorld gw;

   public Game() {
      gw  = new GameWorld();
      gw.initLayout();
      play();
   }

   private void play() {
      // code here to accept and
      // execute user commands that
      // operate on the GameWorld
   }
`
```

Game World Objects

The game world contains a collection which holds two abstract kinds of game objects: "fixed objects" (which are fixed in place) and "moveable objects" (which can move or be moved about the world). For this first version of the game there are three kinds of fixed objects: *pylons*, *fuel cans,* and *oil slicks*; and there are two kinds of moveable objects: *cars* and *birds*. Later we will add other kinds of game objects (both fixed kinds and moveable kinds) as well.

The various game objects have attributes (fields) and behaviors (methods) as defined below. These definitions are requirements which must be properly implemented in your program.

- All game objects have a *location*, defined by floating point non-negative values X and Y. The point (X,Y) is the center of the object. The origin of the "world" (location (0,0)) is the lower left hand corner; you may assume for now that the world is 1000x1000 (although we are going to change this later). All game objects provide the ability for external code to obtain their location *in the form of a single returned object*. Some game objects provide the ability to have their location changed, while others have a location which cannot be changed once the object is created.

- All game objects have a *color*, defined by a value of Java type *java.awt.Color*. All game objects provide the ability for external code to obtain their color in the form of a single returned object. By default, game objects provide the ability to have their color *changed*, unless it is explicitly stated that a certain type of game object has a color which cannot be changed once it is created.

- There are two abstract kinds of game objects; some have changeable locations (that is, they are *moveable*), and others have locations that are not allowed to be changed (that is, they are *fixed*).

- *Pylons* are fixed game objects that have attributes *radius* and *sequenceNumber*. Each pylon is a numbered circular marker that acts as a "waypoint" on the track; following the track is accomplished by driving over the top of pylons in sequential order. (Think of the

track as being of the “off-road, open-country” type rather than a fixed piece of asphalt; the pylons mark the course of the track through the open country. See the diagram below). Pylons are not allowed to change color once they are created.

- *Oil slicks* are fixed game objects that have attributes *width* and *length* (since our game will be 2D, these attributes will represent the width and height of the object on the screen). Hitting an oil slick affects how a car handles, as described below.

- *Fuel Cans* are fixed game objects that have an attribute *size.* The size of a fuel can corresponds to the amount of fuel it contains. Cars that are running low on fuel must pick up a fuel can before they run out of fuel; otherwise they cannot move.

- Moveable game objects have integer attributes *heading* and *speed*. Telling a moveable object to *move()* causes the object to update its location based on its current heading and speed. *Heading* is specified by a *compass angle* in degrees: 0 means heading north (upwards on the screen), 90 means heading east (rightward on the screen), etc.

- Some movable game objects are *steerable*, meaning that they provide an interface that allows other objects to *change* their heading (direction of movement) after they have been created. Note that the difference between *steerable* and *moveable* is that other objects can *request a change in the heading* of *steerable* objects whereas other objects can only request that a *movable* object update its own location according to its current speed and heading.

- *Cars* are moveable, steerable game objects with attributes *width, length, steeringDirection, maximumSpeed, fuelLevel* and *damageLevel*. The *steeringDirection* of a car indicates how the steering wheel is turned (or, equivalently, how the front wheels are oriented) in relation to the front of the car. That is, the steering direction of a car indicates the change the driver would *like* to apply to the *heading* of the car. (Whether or not the steering direction actually *does* get applied – that is, whether turning the steering wheel actually changes the heading along which the car is moving – depends on whether the car currently has fuel and *traction*. Cars in an oil slick, for example, have no traction.) The steering mechanism in a car is such that the steering direction can only be changed in units of 5 degrees at a time, and only up to a maximum of 40 degrees left or right of “straight ahead”.

  The *maximumSpeed* of a car is the upper limit of its *speed* attribute; attempts to accelerate a car beyond its *maximumSpeed* are to be ignored (that is, a car can never go faster than its *maximumSpeed*). Note that different cars may have different *maximumSpeed* values, although initially they all start out with the same value.

  The *fuelLevel* of a car indicates how much fuel it has left; cars with no fuel cannot move.

  The *damageLevel* of a car starts at zero and increases each time the car collides with another car or a bird (see below). The program should define an upper limit on the “damage” a car can sustain. Damage level affects the performance of a car as follows: a car with zero damage can accelerate all the way up to its *maximumSpeed*; cars with the maximum amount of damage cannot move at all; and cars with damage between zero and the maximum damage should be limited in speed to a corresponding percentage of their speed range (for example, a car with 50% of the maximum damage level can only achieve 50% of its maximum speed). When a car incurs damage because it is involved in a collision (see below), its speed is reduced (if necessary) so that this speed-limitation rule is enforced.

- *Birds* are movable game objects with fixed (unchangeable) color which occasionally fly over the racetrack. Birds have an attribute *size* which indicates how big the bird is. Since birds are moveable but not steerable, they always fly in a straight line. If a bird flies directly over a car it causes damage to the car; the damage caused by a bird is ½ the damage caused by colliding with another car but otherwise affects the performance of the car in the same way as described above.

The preceding paragraphs imply several *associations* between classes: an inheritance hierarchy, interfaces such as for *steerable* objects, and aggregation associations between objects and where they are held. You are to develop a UML diagram for the relationships, and then implement it in a Java program. Appropriate use of encapsulation, inheritance, aggregation, abstract classes, and interfaces are important grading criteria. Note that an additional important criterion is that *another programmer must not be able to misuse your classes*; that is, objects created from your classes must be guaranteed to be in a consistent state and must enforce all specified constraints.

Game Play

When the game starts the player has three "lives" (chances to race). The game has a *clock* which counts up starting from zero; in this first version of the game the objective is to drive around the track to the finish line (the last pylon) in the minimum amount of time. (Later we will add other cars and the objective will also include beating the other cars to the finish line.)

The player uses keystroke commands to turn the car's steering wheel; this can cause the car's heading to change (turning the steering wheel only effects the car's heading under certain conditions; see below). The car moves one unit at its current speed in the direction it is currently heading each time the game clock "ticks" (see below).

The car starts out at the first pylon (#1). The player must drive the car so that it intersects the pylons in increasing numerical order. Each time the car reaches the next higher-numbered pylon the player (car) is deemed to have successfully driven that far along the track. Intersecting pylons out of order (that is, reaching a pylon whose number is *more than* one greater than the most recently reached pylon, or whose number is less than or equal to the most recently reached pylon) has no effect on the game.

The fuel level of the car continually goes down as the game continues (fuel level goes down even if the car is not moving; the engine is continually running). If the car's fuel level reaches zero it can no longer move. The player must therefore occasionally drive the car off the track to pick up (intersect with) a fuel can, which has the effect of increasing the car's fuel level by the amount (size) of the fuel can.

Collisions with other cars or with birds cause damage to the car; if the car sustains too much damage it can no longer move. A race is over (the player "loses a life") if the car can no longer move. When the player loses all three lives the game ends.

The program keeps track of several "game state" values: current clock time, lives remaining, last pylon reached, and the current fuel level in the car. Note that these values are part of the "data model" and therefore belong in the GameWorld class.

Commands

Once the game world has been created and initialized, the Game constructor is to call a method name **`play()`** to actually begin the game. **`play()`** repeatedly calls a method named

`getCommand()`, which prompts the user for one- or two-character *commands*. Commands should be input using the Java InputStreamReader, BufferedReader, or Java Scanner class (see the Appendix on "Java Coding Notes", and also page 23 in the text).

The allowable input commands and their meanings are defined below. Each command returned by **`getCommand()`** should invoke a corresponding function in the game world, as follows (note that commands are case sensitive):

- 'a' – tell the game world to **a**ccelerate (increase the speed of) the player's car by a small amount. Note that the effect of acceleration is to be limited based on *damage level*, *fuel level*, and *maximum speed* as described above. Also, cars that are currently in an oil slick have no traction and therefore cannot change their speed.

- 'b' – tell the game world to **b**rake (reduce the speed of) the player's car by a small amount. However, cars that are currently in an oil slick have no traction and cannot change their speed. Note that the minimum speed for a car is zero.

- 'l' (the letter "ell") – tell the game world to change the *steering direction* of the player's car by 5 degrees to the **l**eft (in the negative direction on the compass). Note that this changes the direction of the car's *steering wheel* (or, equivalently, the car's front tires); it does *not* directly (immediately) affect the car's *heading*. See the "tick" command, below.

- 'r' – tell the game world to change the *steering direction* of the player's car by 5 degrees to the **r**ight (in the positive direction on the compass). As above, this changes the direction of the car's *steering wheel*, not the car's *heading*.

- 'o' (the letter "oh") – tell the game world to add a new **o**il slick to the world. Oil slicks get added at a randomly-chosen location and have a randomly-chosen size.

- 'c' – pretend that the player's car has **c**ollided with some other car; tell the game world that this collision has occurred.[1] (For this version of the program we won't actually have any other cars in the simulation, but we need to provide for testing the effect of such collisions.) Colliding with another car increases the damage level of the player's car; if the damage results in the player's car not being able to move then the race is over (the player loses a life).

- 'p*xx*' – pretend that the player's car has collided with (driven over) **p**ylon number ***xx***; tell the game world that this collision has occurred. The effect of driving over a pylon is to check to see whether the number ***xx*** is exactly one greater than the most recent pylon which the car drove over and if so to record *in the car* the fact that the car has now reached the next sequential pylon on the race course. Note that unlike most other commands this command can have one or two digits following the initial character of the command.

- 'f' – pretend that the player's car has collided with (picked up) a **f**uel can; tell the game world that this collision has occurred. The effect of picking up a fuel can is to

[1] In later assignments we will see how to actually detect on-screen collisions such as this; for now we are simply relying on the user to tell the program via a command when collisions have occurred. Inputting a collision command is deemed to be a statement that the collision occurred; it does not matter where objects involved in the collision actually happen to be for this version of the game as long as they exist in the game.

increase the car's fuel level by the size of the fuel can, remove the fuel can from the game, and add a new fuel can with randomly-specified values back into the game.

- 'g' – pretend that a bird has flown over (collided with, ***g****ummed up*) the player's car. The effect of colliding with a bird is to increase the damage to the car as described above under the description of *Birds*.

- 'e' – tell the game world that the player's car just **e**ntered an oil slick. The game world in turn should set a flag in the player's car indicating it is in an oil slick.

- 'x' – tell the game world that the player's car just e**x**ited (is no longer in) an oil slick. Exiting an oil slick should clear the flag in the player's car.

- 'n' – tell the game world to generate **n**ew colors for all objects whose definition allows them to have their color changed. The effect of this command is to process every game object and attempt to assign a new randomly-generated color to it. However, only those objects which are not prohibited from being allowed to have their color changed should be affected.

- 't': tell the game world that the "game clock" has **t**icked. A clock tick in the game world has the following effects: (1) if the player's car is *not* in an oil slick, then the car's heading should be incremented or decremented by the car's *steeringDirection* (that is, the steering wheel turns the car). Note that if the car *is* in an oil slick, the steering wheel has no effect on the car's heading. (2) the car's fuel level is reduced by a small amount. (3) all moveable objects are told to update their positions according to their current heading and speed, and (4) the elapsed time "game clock" is incremented by one (the game clock for this assignment is simply a variable which increments with each tick).

- 'd': generate a **d**isplay by outputting lines of text on the console describing the current game state values. The display should include (1) the number of lives left, (2) the current clock value (elapsed time), (3) the highest pylon number the car has reached sequentially so far, and (4) the car's current fuel level and damage level. All output should be appropriately labeled in easily readable format.

- 'm': tell the game world to output a "**m**ap" showing the current world (see below).

- 'q': call the method `System.exit(0)` to **q**uit the program. Your program should confirm the user's intent to quit before actually exiting.

Additional Details

- The program you are to write is an example of what is called a *discrete simulation*. In such a program there are two basic notions: the ability to *change the simulation state*, and the ability to *advance simulation time*. Changing the state of the simulation has no effect (other than to change the specified values) *until the simulation time is advanced.* In other words, entering commands to change (for example) the car's heading and speed will not actually take effect (that is, will not change the car's *location*) until you enter a "tick" command to advance the *time*. On the other hand, entering a *display* or *map* command after changing state values *will* show the *new* state values even before a tick is entered. You should verify that your program operates like this.

- The code to perform each command must be encapsulated in a separate method (this is because we will be moving it in later assignments). When the *Game* gets a command which requires manipulating the *GameWorld*, the Game must invoke a method in the GameWorld to perform the manipulation (in other words, it is not appropriate for the Game class to be directly manipulating objects in the GameWorld; it must do so by calling an appropriate GameWorld method).

- Any undefined or illegal input should generate an appropriate error message on the console and ignore the input. For example, most input lines with more than a single character are illegal, as are commands with undefined meaning).

- Method ***initLayout()*** is responsible for creating the initial state of the world. This should include adding to the game world at least the following: a minimum of four *Pylon* objects, positioned as you choose and numbered sequentially defining the race course (you may add more than four initial pylons if you like); one *Car*, positioned at the #1 pylon with initial heading and speed both zero; at least two *Bird* objects, randomly positioned with a randomly-generated heading and a speed randomly-chosen from a small range of speeds; at least two *OilSlick* objects with random location and with random width and length chosen from a small range of values; and at least two *FuelCan* objects with random location and with random sizes chosen from a small range. All object initial attributes, including those who values is not otherwise explicitly specified above, should be assigned such that all aspects of the gameplay can be easily tested (for example, Birds should not fly so fast that it is impossible for them to ever cause damage to a Car).

- All classes must be designed and implemented following the guidelines discussed in class, including:
  - *All data fields must be private.*
  - *Accessors / mutators must be provided, but only where the design requires them.*

- It is *not* a requirement in this assignment that single keystrokes invoke action -- the human hits "enter" after typing each key command (we'll see how to fix this in a later assignment).

- Your program must be contained in a Java *package* named "**`a1`**" ("a-one", lower-case). Specifically, every class in your program must be defined in a **`.java`** file which has a "package" statement (for example, "**`package a1;`**") as its first statement. See pages 16-18 and 29-31 in the text for more on the use of Java packages.

- It must be possible to execute the program from a command prompt by changing to the directory containing the **`a1`** package and typing the command: "**`java a1.Starter`**". *Verify for yourself that this works correctly* from a command prompt before submitting your program (see "Deliverables", below).

- Use of subpackages below "package a1" is encouraged. For example, you might create a package "a1.gameObjects" holding the classes comprising the GameObject hierarchy.

- It is a requirement to follow standard Java coding conventions:
  - *class names always start with an upper case letter,*
  - *variable names always start with a lower case letter,*
  - *compound parts of compound names are capitalized (e.g., myExampleVariable),*
  - *Java interface names should start with the letter "I" (e.g., ISteerable).*

- Moving objects need to determine their new location when their *move()* method is invoked, at each time tick. For objects that move in a straight line, the new location can be computed as follows:

  `newLocation(x,y) = oldLocation(x,y) + (deltaX, deltaY),` where
  `deltaX = cos(θ)*speed,`
  `deltaY = sin(θ)*speed,`

  and where **`θ`** = 90 - heading (90 minus the heading). There is a diagram in course notes (p. 228) showing the derivation of these calculations for an arbitrary amount of time; in this assignment we are assuming "time" is fixed at one unit per "tick", so "elapsed time" is 1.

- The program is not required to have any code which actually checks for collisions between objects; that's something we'll add later on. For now, the program simply relies on the user to say when a collision has occurred, using the appropriate command.

- For this assignment <u>all</u> output will be in text form on the console; no "graphical" output is required. The "map" (generated by the '**m**' command) will simply be a set of lines describing the objects currently in the world, similar to the following:

```
Pylon: loc=200,200 color=[64,64,64] radius=10 seqNum=1
Pylon: loc=200,800 color=[64,64,64] radius=10 seqNum=2
Pylon: loc=700,800 color=[64,64,64] radius=10 seqNum=3
Pylon: loc=900,400 color=[64,64,64] radius=10 seqNum=4
Car: loc=180,450 color=[240,0,0] heading=355 speed=50 width=20 length=40
     maxSpeed=50 steeringDirection=5 fuelLevel=5 damage=2
Bird: loc=70,70 color=[255,0,255] heading=45 speed=5 size=15
Bird: loc=950,950 color=[0,255,0] heading=225 speed=10 size=20
OilSlick: loc=250,600 color=[0,0,0] width=10 length=30
OilSlick: loc=800,550 color=[0,0,0] width=20 length=25
FuelCan: loc=350,350 color=[0,0,255] size=3
FuelCan: loc=900,700 color=[0,255,0] size=6
```

  Note that the above "map" describes the game shortly after it has started; the car has moved northward from its initial position at Pylon #1 and has a current heading of 355 (i.e. not quite true north), the car is traveling at its maximum speed, the driver is trying to apply a 5-degree right turn, and so forth. Note also that the appropriate mechanism for implementing this output is to override the **`toString()`** method in each concrete game object class so that it returns a String describing itself. See the attached "Java Coding Notes" for additional details.

  Note that some of the values displayed in the above "map" are displayed as integers whereas the assignment requires that they be maintained in the program as real (floating point) values. This is not a conflict; the program must use real (type *float* or *double*) for these values, but it is acceptable to display them as integers when the real value is a whole number; it is also acceptable to round them to the nearest tenth. See the Java class **`DecimalFormat`** for details on how to do this.

  For this assignment, *the only required depiction of the world is the text output map as shown above*. Later we will learn how to draw a *graphical* depiction of the world, which then for the above map might look something like the image shown below (the pictures in the image are not all precisely to scale, and note that this is only to give you an idea of

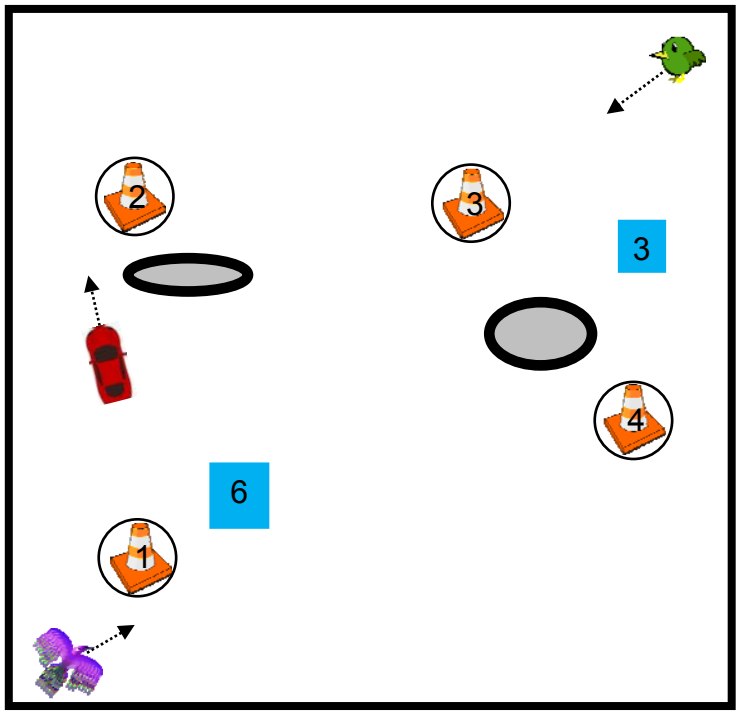
what the above map represents – your program is *not* required to produce any graphical output like this).

- You are not required to use any particular data structure to store the game world objects. However, your program must be able to handle changeable numbers of objects at runtime; this means you can't use a fixed-size array, and you can't use individual variables. Consider either the Java *ArrayList* or *Vector* class for implementing this storage. Note that Java Vectors (and ArrayLists) hold elements of type "Object", but you will need to be able to treat the Objects differently depending on the type of object. You can use the "**`instanceof`**" operator to determine the type of a given Object, but be sure to use it in a polymorphically-safe way. For example, you can write a loop which runs through all the elements of a world Vector and processes each "movable" object with code like:

```
for (int i=0; i<theWorldVector.size(); i++) {
      if (theWorldVector.elementAt(i) instanceof Movable) {
            Movable mObj = (Movable)theWorldVector.elementAt(i);
            mObj.move();
      }
}
```

- It is a requirement for *all* programs in this class that the source code contain *documentation*, in the form of comments explaining what the program is doing, including comments describing the purpose and organization of each class and comments outlining each of the main steps in the code. Points will be deducted for poorly or incompletely documented programs. Use of JavaDoc-style comments is highly encouraged.

- The first step you should undertake for the assignment (after doing the assigned reading in the book) is to develop a *detailed UML diagram* showing the inheritance and aggregation relationships among the required classes, including the fields and methods required. Students are encouraged to ask questions or solicit advice from the instructor outside of class – but have your UML diagram ready – it is the first thing the instructor will ask to see.



Deliverables

There are *three steps* which are required for submitting your program, as follows:

1. Create a *single* file in "ZIP" format containing (1) your UML diagram in .PDF format, (2) the *Java source code* for all the classes in your program, and (3) the compiled (".class") files for your program. Be sure your ZIP file contains the proper subpackage hierarchy. You may use any tool to create the ZIP file as long as it constructs a properly-structured file in ZIP format.

2. *Verify your submission file.* To do this, copy your ZIP file to an empty folder on your machine, unzip it, open a command prompt window, change to the folder where you unzipped the file, and type the command "**java a1.Starter"**. If this does not properly execute your program, your ZIP file is not properly structured; go back to step (1) and fix it. Substantial penalties will be applied to submissions which have not been properly verified according to the above steps.

3. Login to ***SacCT***, select "Assignment 1", and upload your *verified* ZIP file. Also, be sure to take note of the requirement stated in the course syllabus for keeping a *backup copy* of all submitted work (save a copy of your ZIP file).

*All submitted work must be **strictly your own**!*

Appendix – Java Coding Notes

Input Commands

In Java 5.0 and higher, the **Scanner** class will get a line of text from the keyboard:

```
Scanner in = new Scanner (System.in);
System.out.print ("Input some text:");
String line = in.nextLine();
```
or
```
int aValue = in.nextInt();
```

Random Number Generation

The class used to create random numbers in Java is **java.util.Random**. This class contains methods **nextInt()**, which returns a random integer from the entire range of integers, **nextInt(int)**, which returns a random number between zero (inclusive) and the specified integer parameter (exclusive), and **nextBoolean()**, which returns a random boolean value (either true or false). The Random class is discussed on pg. 28 of the text.

Output Strings

The Java routine **System.out.println()** can be used to display text. It accepts a parameter of type String, which can be concatenated from several strings using the "+" operator. If you include a variable which is not a String, it will convert it to a String by invoking its *toString()* method. For example, the following statements print out "The value of I is 3":

```
int i = 3 ;
System.out.println ("The value of I is " + i);
```

Every Java class provides a *toString()* method. Sometimes the result is descriptive; for example, an object of type **java.awt.Color** returns a description of the color. However, if the *toString()* method is the default one inherited from Object, it isn't very descriptive. Your own classes should override *toString()* and provide their own String descriptions – including the *toString()* output provided by the parent class if that class was also implemented by you.

For example, suppose there is a class **Ball** with attribute *radius*, and a subclass of **Ball** named **ColoredBall** with attribute *myColor* of type java.awt.Color. An appropriate *toSring()* method in **ColoredBall** might return a description of a colored ball as follows:

```
public String toString() {
      String parentDesc = super.toString();
      String myDesc = "ColoredBall: " + myColor.toString();
      return parentDesc + myDesc ;
}
```

A program containing a **ColoredBall** called "**myBall**" could then display it as follows:

```
System.out.println ("myBall = " + myBall.toString());
```

or simply:

```
System.out.println ("myBall = " + myBall);
```

JC/PMO:jc
2/4/15