Simulations & library classes

- HW3: RouletteWheel, RouletteGame, RouletteTester
- javadoc
- java.lang classes: String, Character, Integer
- java.util.Random
- for loop

Simulations

programs are often used to model real-world systems
- often simpler/cheaper to study a model
- easier to experiment, by varying parameters and observing the results

- dot race is a simple simulation
  utilized Die object to simulate random steps of each dot

HW3: you will use & develop classes that simulate different betting strategies for roulette
- if you start with 100 credits and play N spins of the wheel, what is your best strategy for winning?
- how do we define winning? ending the N rounds with a profit? ending the N rounds with the most profit?
HW3 classes

- you will be given the RouletteWheel class that models a wheel
- given its specifications, you will write the RouletteGame class that uses RouletteWheel to simulate a game with betting
- you will be given the RouletteTester class that uses RouletteGame to perform repeated simulations and display stats
- finally, you will modify RouletteTester to perform Martingale betting & analyze its effectiveness

recall: Dies is used by lots of other classes
Abstraction & classes

note that RouletteGame will depend upon RouletteWheel

• but you don't have to know the details of how that class works
• can abstract away the details and focus on its behavior

• the javadoc program generates HTML documentation directly from your /** ... */ comments

• viewable in BlueJ by selecting Documentation instead of Source Code at the upper right

• be sure to include javadoc comments for each class (incl. @author & @version) and for every method (incl. @param & @return)

Javadoc specifications

javadoc also provides a way of specifying behavior

• you are given the javadoc HTML specification of the RouletteGame class

• you must implement the specification exactly

• the specification does not dictate internal workings, only external behavior

• since RouletteTester assumes this specification, you must meet it exactly or they will not integrate
Java Standard Library

the Java language provides an extensive library of nearly 4,000 classes
- documentation for the entire library is accessible in javadoc form
- to view in BlueJ, select Java Class Libraries under the Help menu

String class

one of the most useful library classes is String
- technically, its full name is java.lang.String, but the java.lang prefix can be omitted
- a String object encapsulates a sequence of characters and many useful methods
- since Strings are so common, Java provides shortcuts to make them easier to use (and look like primitives)

```
String str = "foo";  \-> String str = new String("foo");
```
- we have already seen a few basic operations on Strings
  you can display Strings using System.out.print and System.out.println

```
System.out.println(firstName);
```
- the ‘+’ operator concatenates two strings (or string and number) together

```
String str = "foo" + "lish" + 1;
```
String javadoc

many of the javadoc details will be revealed later

- for now, important feature is that we can scan the constructors & methods of a class
- String has MANY of each
- can click on a constructor/method link to see more details

Common String methods

```
int length() returns number of chars in String

char charAt(int index) returns the character at the specified index
                        (indices range from 0 to str.length()-1)

boolean contains(String str) returns true if str occurs in the String, else false

int indexOf(char ch) returns index where ch/str first occurs in the String (-1 if not found)

int indexOf(String str) returns the substring from indices start to (end-1)

String toUpperCase() returns copy of String with all letters uppercase

String toLowerCase() returns copy of String with all letters lowercase

boolean equals(String other) returns true if other String has same value
                             returns -1 if less than other String,
                             0 if equal to other String,
                             1 if greater than other String
```
Character class

recall: in Java, strings and characters are different types

String is a class, char is a primitive type

the (java.lang.) Character class has numerous static methods for manipulating characters

- char toLowerCase(char ch) returns lowercase copy of ch
- char toUpperCase(char ch) returns uppercase copy of ch
- boolean isLetter(char ch) returns true if ch is a letter
- boolean isLowerCase(char ch) returns true if lowercase letter
- boolean isUpperCase(char ch) returns true if uppercase letter

String/Character examples

```java
public boolean isVowel(char ch) {
    String vowels = "aeiouAEIOU";
    return vowels.contains(""+ch);
}

public char randomChar(String str) {
    Die d = new Die(str.length());
    return str.charAt(d.roll()-1);
}

public String capitalize(String str) {
    if (str.length() == 0) {
        return str;
    } else {
        return Character.toUpperCase(str.charAt(0)) +
                str.substring(1, str.length());
    }
}
```
Comparing strings

Comparison operators ( < <= > >= ) are defined for primitives but not objects

```java
String str1 = "foo", str2 = "bar";
if (str1 < str2) ... // ILLEGAL
```

== and != are defined for objects, but don't do what you think

```java
if (str1 == str2) ... // TESTS WHETHER THEY ARE THE 
// SAME OBJECT, NOT WHETHER THEY 
// HAVE THE SAME VALUE!
```

Strings are comparable using the equals and compareTo methods

```java
if (str1.equals(str2)) ... // true IF THEY REPRESENT THE 
// SAME STRING VALUE
if (str1.compareTo(str2) < 0) ... // RETURNS -1 if str1 < str2 
// RETURNS  0 if str1 == str2 
// RETURNS  1 if str1 > str2
```

Comparison example

suppose we wanted to compare two names to see which comes first alphabetically

- Kelly Jones < Kelly Miller < Chris Smith < Pat Smith

```java
public void compareNames(String myFirst, String myLast, 
String yourFirst, String yourLast) {
    int lastCompare = myLast.compareTo(yourLast);
    int firstCompare = myFirst.compareTo(yourFirst);
    if (lastCompare < 0 || (lastCompare == 0 && firstCompare < 0)) {
        System.out.println("My name comes before yours alphabetically!");
    } else if (lastCompare > 0 || (lastCompare == 0 && firstCompare > 0)) {
        System.out.println("Your name comes before mine alphabetically!");
    } else {
        System.out.println("We have the same name!");
    }
}
```

note: we have been using == to compare Strings up to this point

- dangerous -- sometimes it works, sometimes it doesn't!
- from now on, always use .equals for Strings
RouletteWheel implementation

- we can use a Die object to choose between 38 values
- the spin method must return a String, since we may want to differentiate between 0 and 00
  - 38 → "00", 37 → ")"
  - 1-36 are converted to a string by concatenating with "":
    e.g., ""+36 → ""+"36" → "36"
- the Die field already keeps track of the number of rolls/spins

```java
public class RouletteWheel {
    private Die roller;

    public RouletteWheel() {
        this.roller = new Die(38);
    }

    public String spin() {
        int number = this.roller.roll();
        if (number == 38) {
            return "00";
        } else if (number == 37) {
            return "0";
        } else {
            return "+"+number;
        }
    }

    public int getNumberOfSpins() {
        return this.roller.getNumRolls();
    }

    public String getColor(String slotValue) {
        String redNums = " 1 3 5 7 9 12 14 16 18 19 21 23 25 27 30 32 34 36 ";
        String blackNums = " 2 4 6 8 10 11 13 15 17 20 22 24 26 28 29 31 33 35 ";
        if (redNums.contains(" "+slotValue+" ")) {
            return "red";
        } else if (blackNums.contains(" "+slotValue+" ")) {
            return "black";
        } else {
            return "green";
        }
    }
}
```

RouletteWheel implementation (cont.)

- getting the color associated with a number is not obvious
  - we could have a large cascading if else to map each number to a color
- instead can use the String method `contains`
  - `redNums` & `blackNums` contain all the numbers of each color, with spaces
    to see if "3" is red/black, see if `redNums`/`blackNums` contains " 3 "

```java
    public String getColor(String slotValue) {
        String redNums = " 1 3 5 7 9 12 14 16 18 19 21 23 25 27 30 32 34 36 ";
        String blackNums = " 2 4 6 8 10 11 13 15 17 20 22 24 26 28 29 31 33 35 ";
        if (redNums.contains(" "+slotValue+" ")) {
            return "red";
        }
        else if (blackNums.contains(" "+slotValue+" ")) {
            return "black";
        }
        else {
            return "green";
        }
    }
```

RouletteWheel implementation (cont.)

- similarly, could have a large cascading if else to map each number to odd/even
- or, could use `charAt` to access the last digit, see if it is "0", "2", "4", "6", or "8"
- instead, the `java.lang.Integer` class contains a static method for converting a string of digits into an int: `parseInt`
  e.g., `Integer.parseInt("14")` → 14

```java
public class RouletteWheel {
    ...
    public String getColor(String slotValue) {
        int numVal = Integer.parseInt(slotValue);
        if (numVal == 0) {
            return "zero";
        } else if (numVal % 2 == 0) {
            return "even";
        } else {
            return "odd";
        }
    }
}
```

Random class

instead of the Die class, we could have used the `java.util.Random` class
- the `nextInt` method returns a random int from 0..(parameter-1)
- other methods can be used to generate different random numbers
RouletteWheel w/ Random

- classes defined within the java.lang library are automatically available
  java.lang.Math
  java.lang.System
  java.lang.String
  java.lang.Character
  java.lang.Integer
- for other libraries, must explicitly import the class

import java.util.Random;
public class RouletteWheel {
    private Random randGen;
    public RouletteWheel() {
        this.randGen = new Random();
    }
    public String spin() {
        int number = this.randGen.nextInt(38);
        if (number == 37) {
            return "00";
        } else {
            return "+"+number;
        }
    }
    public int getNumberOfSpins() {
        // SAME AS BEFORE
    }
    public String getColor(String slotValue) {
        // SAME AS BEFORE
    }
    public String getParity(String slotValue) {
        // SAME AS BEFORE
    }
}

Logic-driven vs. counter-driven loops

sometimes, the number of repetitions is unpredictable
- loop depends on some logical condition, e.g., roll dice until 7 is obtained

often, however, the number of repetitions is known ahead of time
- loop depends on a counter, e.g., roll dice 100 times, traverse every char in string

in general (counting up):
int rep = 0;
while (rep < numRolls) {
    System.out.println(d6.roll() + d6.roll());
    rep++;
}
in general (counting down):
int countNum = 10;
while (countNum > 0) {
    System.out.println(countNum);
    countNum--;
}
For loops

since counter-controlled loops are fairly common, Java provides a special notation for representing them

- a for loop combines all of the loop control elements in the head of the loop

```java
int rep = 0;
for (int rep = 0; rep < NUM_REPS; rep++) {
    STATEMENTS_TO_EXECUTE
}
```

execution proceeds exactly as the corresponding while loop

- the advantage of for loops is that the control is separated from the statements to be repeatedly executed
- also, since all control info is listed in the head, much less likely to forget something

**For loop examples:**

```java
int countNum = 10;
while (countNum > 0) {
    System.out.println(countNum);
    countNum--;
}
System.out.println("BLASTOFF!");
```

```java
Die d6 = new Die();
int numRolls = 0;
int count = 0;
while (numRolls < 100) {
    if (d6.roll() + d6.roll() == 7) {
        count++;
    }
    numRolls++;
}
System.out.println(count);
```

```java
String word = "foobar";
String copy = "";
int i = 0;
while (i < word.length()) {
    copy += word.charAt(i);
    i++;
}
```

```java
Die d6 = new Die();
int count = 0;
for (int numRolls = 0; numRolls < 100; numRolls++) {
    if (d6.roll() + d6.roll() == 7) {
        count++;
    }
}
System.out.println(count);
```

```java
String word = "foobar";
String copy = "";
for (int i = 0; i < word.length(); i++) {
    copy += word.charAt(i);
}
```
Variable scope

**Scope**: the section of code in which a variable exists

- for a field, the scope is the entire class definition
- for a parameter, the scope is the entire method
- for a local variable, the scope begins with its declaration & ends at the end of the enclosing block (i.e., right curly brace)

```java
public class DiceStuff {
    private Die d6;
    ...

    public void showSevens(int numReps) {
        int count = 0;
        for (int numRolls = 0; numRolls < numReps; numRolls++) {
            if (d6.roll() + d6.roll() == 7) {
                count++;
            }
        }
        System.out.println(count);
    }
    ...
}
```

since each method defines its own scope, can reuse the same parameter/local variable name in multiple methods

within a method, can have a parameter/local variable with same name as a field (although confusing)
use this. to differentiate

```java
public class RouletteTester {
    private int startAmount;
    private int betAmount;

    public RouletteTester(int startAmt, int betAmt) {
        this.startAmount = startAmt;
        this.betAmount = betAmt;
    }

    public int playRound(int spinsPerRound, String betType) {
        RouletteGame game = new RouletteGame();
        game.addCredits(this.startAmount);
        for (int roundNum = 0; roundNum < spinsPerRound; roundNum++) {
            int nextBet = Math.min(this.betAmount, game.checkCredits());
            game.makeBet(nextBet, betType);
        }
        return game.checkCredits() - this.startAmount;
    }

    public void playStats(int numRounds, int spinsPerRound, String betType) {
        int numLosses = 0;
        int totalLosses = 0;
        for (int roundNum = 0; roundNum < numRounds; roundNum++) {
            int result = this.playRound(spinsPerRound, betType);
            if (result < 0) {
                numLosses++;
                totalLosses -= result;
            }
        }
        System.out.println(numRounds + " rounds, with " + spinsPerRound + " bets per round on " + betType + ":");
        double lossPercent = 100.0*numLosses / numRounds;
        System.out.println(" Loss percentage = " + lossPercent + " ");
        double avgLoss = (double)totalLosses / numRounds;
        System.out.println(" Avg loss per game = " + avgLoss);
    }
}
```

RouletteTester

**Specify the initial # credits & default bet when constructing a RouletteTester**

**playRound** plays a set number of spins, allows color, parity, or number bets

**playStats** plays a set number of rounds, displays statistics