Chapter 13

Some Standard Java Classes

Java provides a plethora of predefined classes that programmers can use to build sophisticated programs. If you have a choice of using a predefined standard class or devising your own custom class that provides the same functionality, choose the standard class. The advantages of using a standard class include:

- The effort to produce the class is eliminated entirely; you can devote more effort to other parts of the application's development.
- If you create your own custom class, you must thoroughly test it to ensure its correctness; a standard class, while not immune to bugs, generally has been subjected to a complete test suite. Standard classes are used by many developers and thus any lurking errors are usually exposed early; your classes are exercised only by your code, and errors may not be manifested immediately.
- Other programmers may need to modify your code to fix errors or extend its functionality. Standard classes are well known and trusted; custom classes, due to their limited exposure, are suspect until they have been thoroughly examined. Standard classes provide well-documented, well-known interfaces; the interfaces of custom classes may be neither.

Software development today is increasingly *component-based*. In the hardware arena, a personal computer is built by assembling

- a motherboard (a circuit board containing sockets for a microprocessor and assorted support chips),
- a processor and its various support chips,
- memory boards,
- a video card,
- an input/output card (serial ports, parallel port, mouse port)
- a disk controller,
- a disk drive,
- a case,
- a keyboard,

- a mouse, and
- a monitor.

The video card is itself a sophisticated piece of hardware containing a video processor chip, memory, and other electronic components. A technician does not need to assemble the card; the card is used as is off the shelf. The video card provides a substantial amount of functionality in a standard package. One video card can be replaced with another card from a different vendor or with another card with different capabilities. The overall computer will work with either card (subject to availability of drivers for the OS).

Software components are used like hardware components. A software system can be built largely by assembling pre-existing software building blocks. A class is a simple software component; more sophisticated software components can consist of collections of collaborating classes. Java's standard classes are analogous to off-the-shelf hardware components.

13.1 Packages

Java includes a multitude of classes. These classes are organized into collections called *packages*. In order to speed compilation and class loading during execution, the compiler normally has access to a small subset of the standard classes. These common classes include `String` and `System` (as in `System.out.println()`) and are located in a package called `java.lang`. Other classes, some of which we consider in this chapter, require a special statement at the top of the source code that uses these classes:

```
import java.util.Random;
```

This `import` statement makes the definition of the `Random` class available to the compiler. The `Random` class is used to make random number generators which are useful for games and some scientific applications. The `Random` class is explored in § 13.6. The `Random` class is part of the `java.util` package of classes that contain utility classes useful for a variety of programming tasks. Access to the definition of `Random` enables the compiler to verify that client code uses `Random` objects properly. Without the `import` statement that compiler will flag as errors any statements that use `Random`.

`Random`, `String`, and `System` are *simple* class names. The full class name includes the package in which the class is a member. The full class names for the above classes are `java.util.Random`, `java.lang.String`, and `java.lang.System`. If full names are used all the time, no `import` statements are required; however, most programmers will import the classes so the shorter simple names can be used within a program. Full names are required when two classes with the same name from different packages are to be used simultaneously.

Multiple `import` statements may used to import as many class definitions as required. If present, these `import` statements must appear before any other lines in the source code, except for comments.

13.2 Class `System`

We have seen the `print()`, `printf()`, and `println()` methods provided by the `out` public class constant of the `System` class. The `System` class also provides a number of other services, one of which is a class method named `currentTimeMillis()`. The method `System.currentTimeMillis()` gets information from the operating system's clock to determine the number of milliseconds since midnight January 1, 1970. By calling the method twice and comparing the two results we can determine the time elapsed between two events. Try the following interactive session:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> long start = System.currentTimeMillis();
> System.currentTimeMillis() - start
30166
```

Type in the first statement and, just before hitting Enter, note the time on a clock. Go ahead an type the second line but wait to hit the Enter key until about 30 seconds has elapsed since last pressing the Enter key (if you are a slow typist, you may want to wait a full minute). The value displayed is the number of milliseconds ($\frac{1}{1000}$ seconds) between the two Enter presses.

Using `System.currentTimeMillis()` interactively to time human user activity is rather limited, but `System.currentTimeMillis()` can be very useful to time program execution and synchronizing program activity with particular time periods. `Stopwatch` (13.1) implements a software stopwatch that we can use to time program execution:

```
public class Stopwatch {
    //  A snapshot of the system time when the stopwatch is started
    private long startTime;
    //  The number of milliseconds elapsed from when the stopwatch was
    //  started to when it was stopped
    private long elapsedTime;
    //  Is the stopwatch currently running?
    private boolean running = false;
    //  Start the watch running
    public void start() {
        running = true;
        startTime = System.currentTimeMillis();
    }
    //  Stop the watch and note elapsed time
    public void stop() {
        elapsedTime = System.currentTimeMillis() - startTime;
        running = false;
    }
    //  Return the elapsed time
    public long elapsed() {
        //  This method should only be called on a stopped watch
        if ( running ) {  //  Issue warning message
            System.out.println("**** Stopwatch must be stopped to "
                               + "provide correct elapsed time.");
        }
        return elapsedTime;
    }
}
```

Listing 13.1: `Stopwatch`—implements a software stopwatch to time code execution

To use `Stopwatch` objects:

1. Create a stopwatch object: `Stopwatch s = new Stopwatch();`

2. At the beginning of an event start it: `s.start();`

3. At the end of the event stop it: `s.stop();`

4. Read the elapsed time: `s.elapsed();`

13.3 Class `String`

We have been using strings so much they may seem like primitive types, but strings are actually `String` objects. A string is a sequence of characters. In Java, a `String` object encapsulates a sequence of Unicode (see `http://www.unicode.org`) characters and provides a number of methods that allow programmers to use strings effectively. Since `String` is a member of the `java.lang` package, no import is necessary.

A *string literal* is a sequence of characters enclosed within quotation marks, as in

```
"abcdef"
```

A string literal is an instance of the `String` class. The statement

```
String w = "abcdef";
```

assigns the `String` object reference variable `w` to the `String` object `"abcdef"`. To make a copy of this literal string, the `new` operator must be used:

```
String w = new String("abcdef");
```

This assignment directs `w` to refer to a distinct object from the literal `"abcdef"` object, but the characters that make up the two strings will be identical.

Table 13.1 lists some of the more commonly used methods of the `String` class.

Some Methods of the `String` Class
`char charAt(int i)` Returns the character at the specified index `i`. An index ranges from 0 to `length() - 1`. The first character of the sequence is at index 0, the next at index 1, and so on.
`int length()` Returns the number of characters in a string
`boolean equals(Object s)` Determines if one string contains exactly the same characters in the same sequence as another string
`String concat(String str)` Splices one string onto the end of another string
`int indexOf(int c)` Returns the position of the first occurrence of a given character within a string
`String toUpperCase()` Returns a new string containing the capitalized version of the characters in a given string. The string used to invoke `toUpperCase()` is not modified.
`String valueOf(int i)` Converts an `int` into its string representation. The method is overloaded for all the primitive types.

Table 13.1: A subset of the methods provided by the `String` class

`StringExample` (13.2) exercises some string methods.

```
public class StringExample {
    public static void test() {
        String str = "abcdef";
        //  Should print 6
        System.out.println(str.length());
        //  Also should print 6
        System.out.println("abcdef".length());
        //  Use the special concatenation operator
        System.out.println(str + "ghijk");
        //  Use the concatenation method
        System.out.println(str.concat("ghijk"));
        //  See the uppercase version of the string
        System.out.println(str.toUpperCase());
        //  Should print "true"
        System.out.println(str.equals("abcdef"));
        //  Should print "[c]"
        System.out.println("[" + str.charAt(2) + "]");
        //  Should print 2
        System.out.println("Character c is in position " + str.indexOf('c'));
    }
}
```

Listing 13.2: `StringExample`—Illustrates some string methods

Some parts of `StringExample` are worth special attention:

- The statement

  ```
  System.out.println("abcdef".length());
  ```

  emphasizes the fact that a string literal in Java is an actual object.

- The `+` operator and `concat()` method are two different ways to concatenate strings in Java. The `String` class is special in Java because no other classes can use an operator to invoke a method. The `+` operator executes the `concat()` method.

- The first character in a string as at position zero, the second character is at position one, and so forth. Figure 13.1 illustrates.

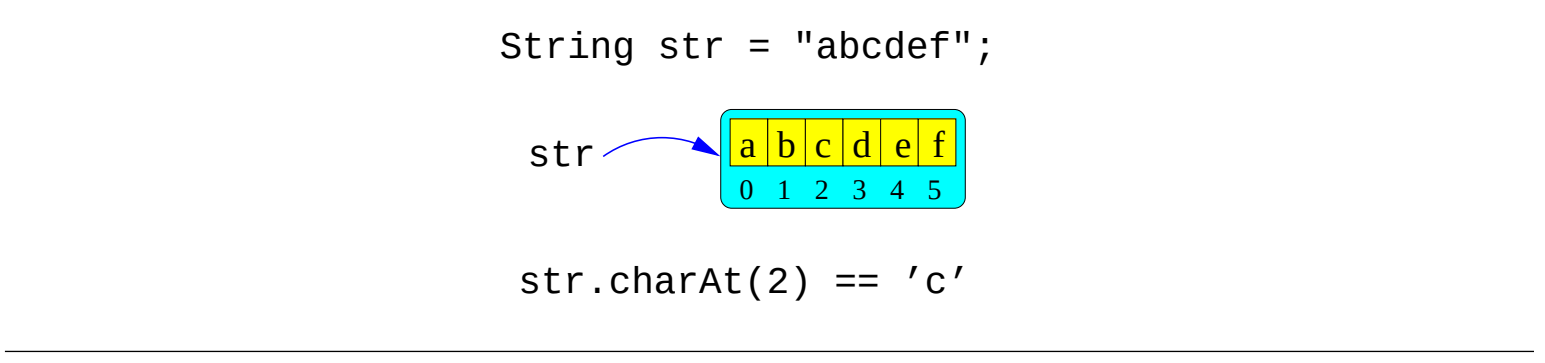

Figure 13.1: String indexing. The character `'c'` is found at index 2.

- Observe that none of the methods used in `StringExample` modify the contents of `str`. The expression

  ```
  str.toUpperCase()
  ```

  returns a new string object with all capital letters; it does *not* modify `str`. In fact, none of the methods in the `String` class modify the string object upon which they are invoked. A string object may *not* be modified after it is created. Such objects are called *immutable*. It is acceptable to modify a reference, as in:

  ```
  str = str.toUpperCase();
  ```

  This statement simply reassigns `str` to refer to the new object returned by the method call. It does not modify the string object to which `str` was originally referring. Figure 13.2 illustrates.

One additional useful `String` method is `format()`. It is a class (`static`) method that uses a format string like `System.out.printf()`. Whereas `System.out.printf()` displays formatted output, `String.format()` instead produces a string, as demonstrated in the following interaction:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> double x = 3.69734;
> x
3.69734
> String.format("%.2f", x)
"3.70"
```


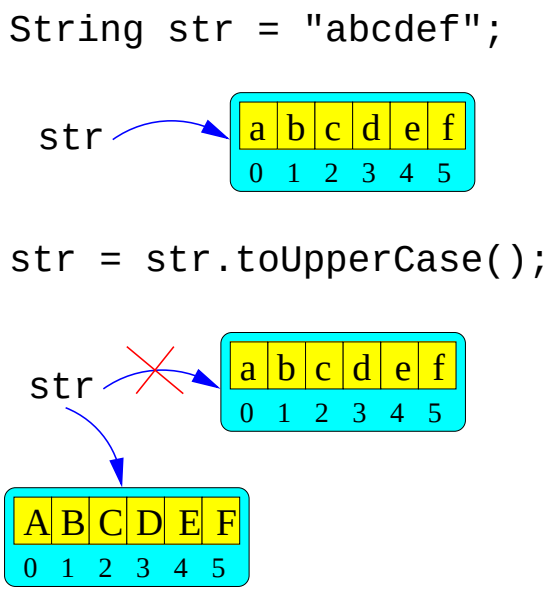

Figure 13.2: String reassignment. The reference is redirected to point to a different string object; the original string object is unaffected (but it may be garbage collected if it is no longer referenced elsewhere).

The `%.2f` control code specifies a floating point number rounded to two decimal places. The control codes used in the `String.format()` format string are exactly those used the `System.out.printf()` format string. This is because the two methods use a common formatting object (of class `java.util.Formatter`) to do their work.

13.4 Primitive Type Wrapper Classes

Java's distinction between primitive types and reference types dilutes its object-orientation. Some OO languages, like Smalltalk, have no primitive types—everything is an object. As we'll discover, it is often convenient to treat a primitive type as if it were an object.

The `java.lang` package provides a "wrapper" class for each primitive type. These wrapper classes serve two roles:

1. They can be used to wrap a primitive type value within an object. This objectifies the primitive value and adds a number of useful methods that can act on that value. The state of the object is defined solely by the value of the primitive type that it wraps.

2. They provide services or information about the primitive type they wrap. For example:

   - The minimum and maximum values that the primitive type can represent are available.
   - Conversions to and from the primitive type's human-readable string representations can be performed.

Table 13.2 lists all of Java's wrapper classes.

We'll focus on the `Integer` class, but the other wrapper classes provide similar functionality for the types they wrap. Some useful `Integer` methods are shown in Table 13.3.

Wrapper Class	Primitive Type
`Byte`	`byte`
`Short`	`short`
`Character`	`char`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Boolean`	`boolean`

Table 13.2: Wrapper classes used to objectify primitive types

Some Methods of the `Integer` Class

`static int parseInt(String s)`
A class method that converts a string representation of an integer to an `int`. For example,
`int value = Integer.parseInt("256");`
Note that the following statement is illegal:
`int value = "256";  //  Illegal!`
since a `String` object reference is not assignment compatible with an `int`.

`static int parseInt(String s, int b)`
This overloaded method works with different number bases. For example,
`int value = Integer.parseInt("101", 2);`
assigns the value five to `value`, since $101_2 = 5_{10}$.

`static Integer valueOf(String s)` Like `parseInt()`, but returns a reference to a new `Integer` object, not a primitive `int` value.
`static Integer valueOf(String s, int b)` As with `parseInt()`, an overloaded version that deals with different number bases.
`int intValue()` Returns the primitive `int` value wrapped by this `Integer` object. Similar methods exist for the other primitive numeric types.
`String toString()` Returns the `String` representation the wrapped `int` value.
`static String toString(int i)` A class method that returns the `String` representation the value of `i`

Table 13.3: Some useful `Integer` methods

Public constant fields `MIN_VALUE` and `MAX_VALUE` store, respectively, the smallest and largest values that can be represented by the `int` type. The `Integer` constructor is overloaded and accepts either a single `int` or single `String` reference as a parameter. `IntegerTest` (13.3) shows how the `Integer` class can be used.

```
public class IntegerTest {
```



```
    public static void main(String[] args) {
        Integer i1 = new Integer("500");  //  Make it from a string
        Integer i2 = new Integer(200);    //  Make it from an int
        int i;

        i = i1.intValue();
        System.out.println("i = " + i);  //  Prints 500
        i = i2.intValue();
        System.out.println("i = " + i);  //  Prints 200
        i = Integer.parseInt("100");
        System.out.println("i = " + i);  //  Prints 100
        i1 = Integer.valueOf("600");
        i = i1.intValue();
        System.out.println("i = " + i);  //  Prints 600
        i = Integer.valueOf("700").intValue();
        System.out.println("i = " + i);  //  Prints 700
    }
}
```

Listing 13.3: `IntegerTest`—Exercises the `Integer` class

A technique called *autoboxing* allows a primitive type to be converted automatically to an object of its wrapper type (*boxing*), and it allows an object of a wrapper type to be converted automatically to its corresponding primitive value (*unboxing*). For example, the compiler converts the following statement

```
Double d = 3.14;  //  3.14 is boxed
```

into

```
Double d = new Double(3.14);   //  How the compiler treats it
```

Similarly, the following code converts the other direction:

```
Double d1 = new Double(10.2);
double d2 = d1;  //  10.2 is unboxed
```

Boxing and unboxing is performed automatically during parameter passing and method returns as well. A method that expects a wrapper type object will accept a compatible primitive type and vice-versa. Autoboxing largely eliminates the need to use explicit constructor calls and methods such as `intValue()`. These methods are present because earlier versions of Java did not support autoboxing.

13.5 Class `Math`

The `Math` class contains mathematical methods that and provide much of the functionality of a scientific calculator. Table 13.4 lists only a few of the available methods.

Some Methods of the `Math` Class
`static double sqrt(double x)` Computes the square root of a number: $\sqrt{x}$ = `sqrt(x)`.
`static double pow(double x, double y)` Raises one number to the power of another: x^y = `pow(x, y)`.
`static double log(double x)` Computes the natural logarithm of a number: $\log_e x = \ln x$ = `log(x)`.
`static double abs(double x)` Returns the absolute value of a number: $\lvert x \rvert$ = `abs(x)`.
`static double cos(double x)` Computes the cosine of a value specified in radians: $\cos \frac{\pi}{2}$ = `cos(PI/2)`; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent.

Table 13.4: Some useful `Math` methods

All the methods in the `Math` class are *class* methods (`static`), so they can be invoked without the need to create a `Math` object. In fact, the `Math` class defines a private constructor, so it is impossible to create a `Math` instance (§ 16.2). Two public class constants are also available:

- `PI`—the `double` value approximating π
- `E`—the `double` value approximating e, the base for natural logarithms.

Figure 13.3 shows a problem that can be solved using methods found in the `Math` class. A point (x,y) is to be rotated about the origin (point $(0,0)$), while the point (p_x, p_y) is fixed. The distance between the moving point and the fixed point at various intervals is to be displayed in a table.

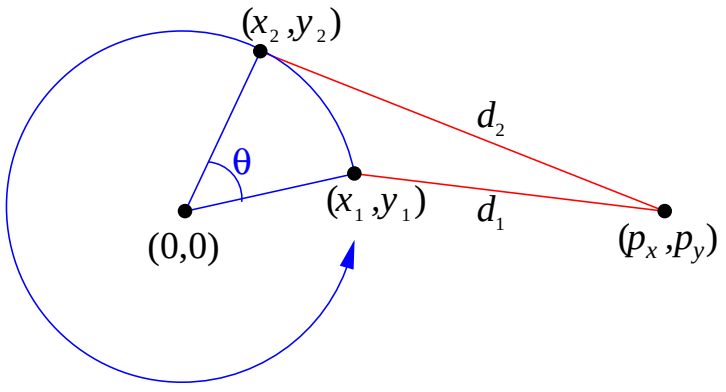

Figure 13.3: A problem that can be solved using methods in the `Math` class.

Point (p_x, p_y) could represent a spacecraft at a fixed location relative to a planet centered at the origin. The moving point could then represent a satellite orbiting the planet. The task is to compute the changing distance between the spacecraft and the satellite as the satellite orbits the planet. The spacecraft is located in the same plane as the satellite's orbit.

Two problems must be solved, and facts from mathematics provide the answers:

1. **Problem**: The location of the moving point must be recomputed as it orbits the origin.

Solution: Given an initial position (x_1, y_1) of the moving point, a rotation of θ degrees around the origin will yield a new point at

$$\begin{aligned} x_2 &= x_1 \cos\theta - y_1 \sin\theta \\ y_2 &= x_1 \sin\theta + y_1 \cos\theta \end{aligned}$$

2. **Problem**: The distance between the moving point and the fixed point must be recalculated as the moving point moves to a new position.

   Solution: The distance d_1 between two points (p_x, p_y) and (x_1, y_1) is given by the formula

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

   The distance d_2 in Figure 13.3 is

$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

`OrbitDistanceTable` (13.4) uses these mathematical results to compute a partial orbit distance table.

```
public class OrbitDistanceTable {
    //  Location of orbting point is (x,y)
    private static double x;
    private static double y;

    //  Location of fixed point is always (100, 0), AKA (fixedX, fixedY)
    private static final double fixedX = 100;
    private static final double fixedY = 0;

    //  Precompute the cosine and sine of 10 degrees
    private static final double COS10 = Math.cos(Math.toRadians(10));
    private static final double SIN10 = Math.sin(Math.toRadians(10));

    public static void move() {
        double xOld = x;
        x = x*COS10 - y*SIN10;
        y = xOld*SIN10 + y*COS10;
    }

    public static double distance(double x1, double y1, double x2, double y2) {
        return Math.sqrt((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));
    }

    public static void report() {
        System.out.printf("%9.3f %9.3f %9.3f %n", x, y,
                          distance(x, y, fixedX, fixedY));
    }

    public static void main(String[] args) {
        //  Initial position of (x,y)
        x = 50;
        y = 0;
        System.out.println("      x          y     Distance");
```



```
        System.out.println("---------------------------------");
        //  Rotate 30 degrees in 10 degree increments
        report();
        move();     //  10 degrees
        report();
        move();     //  20 degrees
        report();
        move();     //  30 degrees
        report();
    }
}
```

Listing 13.4: `OrbitDistanceTable`—Generates a partial table of the orbital position and distance data

13.6 Class Random

Instances of the `java.util.Random` class are used to generate *random numbers*. Random numbers are useful in games and simulations. For example, many board games use a die (one of a pair of dice) to determine how many places a player is to advance. A software adaptation of such a game would need a way to simulate the random roll of a die.

All algorithmic random number generators actually produce *pseudorandom* numbers, not true random numbers. A pseudorandom number generator has a particular period, based on the nature of the algorithm used. If the generator is used long enough, the pattern of numbers produced repeats itself exactly. For example, a particularly poor pseudorandom number generator with a very short period may generate the sequence:

$$2013, 3, 138769, -2342, 7193787, 2013, 3, 138769, -2342, 7193787, \ldots$$

This repetition of identical values disqualifies all algorithmic generators from being true random number generators. The good news is that all practical algorithmic pseudorandom number generators have *much* longer periods.

Java's standard pseudorandom number generator is based on Donald Knuth's widely used linear congruential algorithm [3]. It works quite well for most programming applications requiring random numbers. The algorithm is given a seed value to begin, and a formula is used to produce the next value. The seed value determines the sequence of numbers generated; identical seed values generate identical sequences.

The `Random` class has an overloaded constructor:

- `Random()`—the no parameter version uses the result of `System.currentTimeMillis()` (the number of milliseconds since January 1, 1970) as the seed. Different executions of the program using a `Random` object created with this parameter will generate different sequences since the constructor is called at different times (unless the system clock is set back!).

- `Random(long seed)`—the single parameter version uses the supplied value as the seed. This is useful during development since identical sequences are generated between program runs. When changes are made to a malfunctioning program, reproducible results make debugging easier.

Some useful `Random` methods are shown in Table 13.5.

Some Methods of the `Random` Class
`int nextInt(int n)` Overloaded: • No parameter–returns the next uniformly distributed pseudorandom number in the generator's sequence converted to an integer. • Single positive integer parameter (n)— works like the no parameter version except makes sure the result r is in the range $0 \leq r < n$.
`double nextDouble()` Returns the next uniformly distributed pseudorandom number in the generator's sequence converted to a `double`.
`void setSeed(long seed)` Resets the seed value.

Table 13.5: Some useful `Random` methods

To see how the `Random` class might be used, consider a *die*. A die is a cube containing spots on each of its six faces. The number of spots vary in number from one to six. The word *die* is the singular form of *dice*. Figure 13.4 shows a pair of dice.

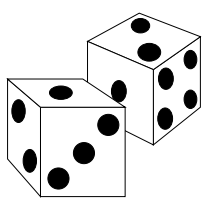

Figure 13.4: A pair of dice. The top face of the left die is one, and the top face of the right die is two.

Dice are often used to determine the number of moves on a board game and are used in other games of chance. A player rolls a die or sometimes a pair of dice, and the side(s) that face up have meaning in the game being played. The value of a face after a roll is determined at random by the complex tumbling of the die.

`DieGame` (13.5) simulates rolling a pair of dice.

```
public class DieGame {
    public static void play() {
        Die die1 = new Die();
        Die die2 = new Die();
        //  First, print the value on the faces
        System.out.println("Initially:");
        die1.show();
        die2.show();
        //  Then, roll the dice
        die1.roll();
        die2.roll();
        //  Finally, look at the results
        System.out.println("After rolling:");
```



```
        die1.show();
        die2.show();
    }
}
```

Listing 13.5: DieGame—simulates rolling a pair of dice

A sample output of the DieGame would look like:

```
Initially:
+-------+
| *     |
|       |
|     * |
+-------+
+-------+
| *   * |
|   *   |
| *   * |
+-------+
After rolling:
+-------+
| * * * |
|       |
| * * * |
+-------+
+-------+
| *   * |
|   *   |
| *   * |
+-------+
```

DieGame uses objects of the Die class (13.6). As DieGame shows, a Die object must:

- be created in a well-defined state, so show() can be invoked before roll() is called,
- respond to a roll() message to possibly change its visible face, and
- be able to show its top face in a text-based graphical fashion (rather than just printing the number of spots).

```
import java.util.Random;

public class Die {
    //  The number of spots on the visible face
    private int value;
    //  The class's pseudorandom number generator
```



```
    private static Random random = new Random();
    public Die() {
        roll();   //  Initial value is random
    }
    //  Sets value to a number in the range 1, 2, 3, ..., 6
    public void roll() {
        value = random.nextInt(6) + 1;
    }
    //  Draws the visible face in text-based graphics
    public void show() {
        System.out.println("+-------+");
        if (value == 1) {
            System.out.println("|       |");
            System.out.println("|   *   |");
            System.out.println("|       |");
        } else if (value == 2) {
            System.out.println("| *     |");
            System.out.println("|       |");
            System.out.println("|     * |");
        } else if (value == 3) {
            System.out.println("|     * |");
            System.out.println("|   *   |");
            System.out.println("| *     |");
        } else if (value == 4) {
            System.out.println("| *   * |");
            System.out.println("|       |");
            System.out.println("| *   * |");
        } else if (value == 5) {
            System.out.println("| *   * |");
            System.out.println("|   *   |");
            System.out.println("| *   * |");
        } else if (value == 6) {
            System.out.println("| * * * |");
            System.out.println("|       |");
            System.out.println("| * * * |");
        } else {  //  Defensive catch all:
            System.out.println(" ***  Error: illegal die value ***");
        }
        System.out.println("+-------+");
    }
}
```

Listing 13.6: `Die`—simulates a die

The client code, `DieGame`, is shielded from the details of the `Random` class. A `Random` object is encapsulated within the `Die` object. Observe that within the `Die` class:

- The `value` field is an instance variable. Each `Die` object must maintain its own state, so `value` cannot be shared (that is, it cannot be a class variable).

- The `random` field is a class variable. The `random` variable references a `Random` object and is used to generate the pseudorandom numbers that represent the visible face after a roll. This object can be shared by all dice, since each call to its `nextInt()` method simply returns the next pseudorandom number in the sequence. The next value of the `value` field for a particular die is not and should not be influenced by the current state of that die. While it is not logically incorrect to make `random` an instance variable, it would be an inefficient use of time and space since

  1. Each time a new `Die` object is created, a new `Random` object must also be created. Object creation is a relatively time-consuming operation.
  2. Each `Random` object associated with each `Die` object occupies memory. If only one `Random` object is really required, creating extra ones wastes memory.

- The `random` field is initialized when declared, so it will be created when the JVM loads the class into memory.

- The constructor automatically initializes `value` to be a random number in the range $1, 2, 3, \ldots, 6$ by calling the `roll()` method. Even though `roll()` contains only one statement that could be reproduced here, this approach avoids duplicating code. In terms of maintainability this approach is better, since if in the future the nature of a roll changes (perhaps a ten-sided die), code only needs to be modified in one place.

  If no constructor were provided, `value` would by default be assigned to zero, not a valid value since all faces have at least one spot on them.

- The statement

```
value = random.nextInt(6) + 1;
```

  assigns a number in the range $1, 2, \ldots, 6$, since the expression `random.nextInt(6)` itself returns a number in the range $0, 1, \ldots, 5$.

13.7 Summary

- Add summary items here.

13.8 Exercises

1. Devise a new type of traffic light that has `TrafficLightModel` (10.1) as its superclass but does something different from `TurnLightModel` (11.1). Test your new class in isolation.
2. Create a new intersection class (subclass of `Intersection` (10.4)) Test your new type of intersection.

