6.10 Case Study: A Game of Chance; Introducing `enum` Types

A popular game of chance is a dice game known as craps, which is played in casinos and back alleys throughout the world. The rules of the game are straightforward:

You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.

Figure 6.8 simulates the game of craps, using methods to implement the game's logic. The `main` method (lines 20–66) calls the `rollDice` method (lines 69–80) as necessary to roll the dice and compute their sum. The sample outputs show winning and losing on the first roll, and winning and losing on a subsequent roll.

```
 6.8: Craps.java
ılates the dice game craps.
ecurity.SecureRandom;
   4
 class Craps {
random number generator for use in method rollDice
nal SecureRandom randomNumbers = new SecureRandom();
   8
 constants that represent the game status
:us {CONTINUE, WON, LOST};
  11
 represent common rolls of the dice
final int SNAKE_EYES = 2;
ic final int TREY = 3;
c final int SEVEN = 7;
 final int YO_LEVEN = 11;
 final int BOX_CARS = 12;
  18
 one game of craps
oid main(String[] args) {
0; // point if no win or loss on first roll
tus; // can contain CONTINUE, WON or LOST
  23
= rollDice(); // first roll of the dice
  25
ame status and point based on first roll
:ch (sumOfDice) {
 // win with 7 on first roll
EN: // win with 11 on first roll
meStatus = Status.WON;
      break;
EYES: // lose with 2 on first roll
// lose with 3 on first roll
RS: // lose with 12 on first roll
eStatus = Status.LOST;
      break;
 did not win or lose, so remember point
us = Status.CONTINUE; // game is not over
= sumOfDice; // remember the point
ut.printf("Point is %d%n", myPoint);
```

```
      break;
     }
 43
 game is not complete
tus == Status.CONTINUE) { // not WON or LOST
 rollDice(); // roll dice again
 47
etermine game status
ce == myPoint) { // win by making point
meStatus = Status.WON;
       }
     else {
fDice == SEVEN) { // lose by rolling 7 before point
meStatus = Status.LOST;
         }
       }
     }
 58
ay won or lost message
tatus == Status.WON) {
t.println("Player wins");
     }
   else {
:.println("Player loses");
     }
66   }
 67
culate sum and display results
tic int rollDice() {
k random die values
 randomNumbers.nextInt(6); // first die roll
 randomNumbers.nextInt(6); // second die roll
 73
 + die2; // sum of die values
 75
y results of this roll
ntf("Player rolled %d + %d = %d%n", die1, die2, sum);
 78
eturn sum;
80   }
```

```
81 }
```

```
Player rolled 5 + 6 = 11
      Player wins
```

```
Player rolled 5 + 4 = 9
      Point is 9
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
      Player wins
```

```
Player rolled 1 + 2 = 3
     Player loses
```

```
Player rolled 2 + 6 = 8
      Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
     Player loses
```

Fig. 6.8

`Craps` class simulates the dice game craps.

Method `rollDice`

In the rules of the game, the player must roll *two* dice on each roll. We declare method `rollDice` (lines 69–80) to roll the dice and compute and print their sum. Method `rollDice` is declared once, but it's called from two places (lines 24 and 46) in `main`, which contains the logic for one complete game of craps. Method `rollDice` takes no arguments, so it has an empty parameter list. Each time it's called, `rollDice` returns the sum of the dice, so the return type `int` is indicated in the method header (line 69). Although lines 71 and 72 look the same (except for the die names), they do not necessarily produce the same result. Each statement produces a *random* value in the range 1–6. Variable `randomNumbers` (used in lines 71–72) is *not* declared in the method. Instead it's declared as a `private static final` variable of the class and initialized in line 7. This enables us to create one `SecureRandom` object that's reused in each call to `rollDice`. If there were a program that contained multiple instances of class `Craps`, they'd all share this one `SecureRandom` object.

Method `main`'s Local Variables

The game is reasonably involved. The player may win or lose on the first roll, or may win or lose on any subsequent roll. Method `main` (lines 20–66) uses

- local variable `myPoint` (line 21) to store the “point” if the player doesn’t win or lose on the first roll,
- local variable `gameStatus` (line 22) to keep track of the overall game status and
- local variable `sumOfDice` (line 24) to hold the sum of the dice for the most recent roll.

Variable `myPoint` is initialized to `0` to ensure that the application will compile. If you do not initialize `myPoint`, the compiler issues an error, because `myPoint` is not assigned a value in *every* `case` of the `switch` statement, and thus the program could try to use `myPoint` before it’s assigned a value. By contrast, `gameStatus` *is* assigned a value in *every* `case` of the `switch` statement (including the `default` case)—thus, it’s guaranteed to be initialized before it’s used, so we do not need to initialize it in line 22.

`enum` Type `Status`

Local variable `gameStatus` (line 22) is declared to be of a new type called `Status` (declared at line 10). Type `Status` is a `private` member of class `Craps`, because `Status` will be used only in that class. `Status` is a type called an `enum` **type**, which, in its simplest form, declares a set of constants represented by identifiers. An `enum` type is a special kind of class that’s introduced by the keyword `enum` and a type name (in this case, `Status`). As with classes, braces delimit an `enum` declaration’s body. Inside the braces is a comma-separated list of `enum` **constants**, each representing a unique

value. The identifiers in an `enum` must be *unique*. You'll learn more about `enum` types in Chapter 8.

Good Programming Practice 6.1

Use only uppercase letters in the names of `enum` *constants to make them stand out and remind you that they're not variables.*

Variables of type `Status` can be assigned only the three constants declared in the `enum` (line 10) or a compilation error will occur. When the game is won, the program sets local variable `gameStatus` to `Status.WON` (lines 30 and 50). When the game is lost, the program sets local variable `gameStatus` to `Status.LOST` (lines 35 and 54). Otherwise, the program sets local variable `gameStatus` to `Status.CONTINUE` (line 38) to indicate that the game is not over and the dice must be rolled again.

Good Programming Practice 6.2

Using `enum` *constants (like* `Status.WON`, `Status.LOST` *and* `Status.CONTINUE`*) rather than literal values (such as*

0, 1 and 2) makes programs easier to read and maintain.

Logic of the `main` Method

Line 24 in `main` calls `rollDice`, which picks two random values from 1 to 6, displays the values of the first die, the second die and their sum, and returns the sum. Method `main` next enters the `switch` statement (lines 27–42), which uses the `sumOfDice` value from line 24 to determine whether the game has been won or lost, or should continue with another roll. The values that result in a win or loss on the first roll are declared as `private static final int` constants in lines 13–17. The identifier names use casino parlance for these sums. These constants, like `enum` constants, are declared by convention with all capital letters, to make them stand out in the program. Lines 28–31 determine whether the player won on the first roll with `SEVEN` (`7`) or `YO_LEVEN` (`11`). Lines 32–36 determine whether the player lost on the first roll with `SNAKE_EYES` (`2`), `TREY` (`3`), or `BOX_CARS` (`12`). After the first roll, if the game is not over, the `default` case (lines 37–41) sets `gameStatus` to `Status.CONTINUE`, saves `sumOfDice` in `myPoint` and displays the point.

If we're still trying to "make our point" (i.e., the game is continuing from a prior roll), lines 45–57 execute. Line 46 rolls the dice again. If `sumOfDice` matches `myPoint` (line 49), line 50 sets `gameStatus` to `Status.WON`, then the loop terminates because the game is complete. If `sumOfDice` is `SEVEN` (line 53), line 54 sets `gameStatus` to

`Status.LOST`, and the loop terminates because the game is complete. When the game completes, lines 60–65 display a message indicating whether the player won or lost, and the program terminates.

The program uses the various program-control mechanisms we've discussed. The `Craps` class uses two methods—`main` and `rollDice` (called twice from `main`)—and the `switch`, `while`, `if`…`else` and nested `if` control statements. Note also the use of multiple `case` labels in the `switch` statement to execute the same statements for sums of `SEVEN` and `YO_LEVEN` (lines 28–29) and for sums of `SNAKE_EYES`, `TREY` and `BOX_CARS` (lines 32–34).

Why Some Constants Are Not Defined as `enum` Constants

You might be wondering why we declared the sums of the dice as `private static final int` constants rather than as `enum` constants. The reason is that the program must compare the `int` variable `sumOfDice` (line 24) to these constants to determine the outcome of each roll. Suppose we declared `enum Sum` containing constants representing the five sums used in the game, then used these constants in the `switch` statement (lines 27–42). Doing so would prevent us from using `sumOfDice` as the `switch` statement's controlling expression, because Java does *not* allow you to compare an

`int` to an `enum` constant. To achieve the same functionality as the current program, we'd have to use a variable `currentSum` of type `Sum` as the `switch`'s controlling expression. Unfortunately, Java does not provide an easy way to convert an `int` value to a particular `enum` constant. This could be done with a separate `switch` statement. This would be cumbersome and would not improve the program's readability (thus defeating the purpose of using an `enum`).