Lesson 5 – Part I
Methods: A Deeper Look

Assoc. Prof. Marenglen Biba

OBJECTIVES

In this Chapter you'll learn:

- How `static` methods and fields are associated with an entire class rather than specific instances of the class.
- To use common `Math` methods available in the Java API.
- To understand the mechanisms for passing information between methods.
- How the method call/return mechanism is supported by the method-call stack and activation records.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

6.1 Introduction

- Best way to develop and maintain a large program is to construct it from small, simple pieces, or modules.
  - divide and conquer.
- Topics in this chapter
  - `static` methods
  - Declare a method with more than one parameter
  - Simulation techniques with random-number generation.
  - How to declare values that cannot change (i.e., constants) in your programs.
  - Method overloading.

6.2 Program Modules in Java

- Java programs combine new methods and classes that you write with predefined methods and classes available in the Java Application Programming Interface and in other class libraries.
- Related classes are typically grouped into packages so that they can be imported into programs and reused.
- Example: Create packages in Netbeans

6.2 Program Modules in Java (Cont.)

- Methods help you modularize a program by separating its tasks into self-contained units.
- Statements in method bodies
  - Written only once
  - Hidden from other methods
  - Can be reused from several locations in a program
- Divide-and-conquer approach
  - Constructing programs from small, simple pieces
- Software reusability
  - Use existing methods as building blocks to create new programs.
- Dividing a program into meaningful methods makes the program easier to debug and maintain.

Software Engineering Observation 6.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively.

Error-Prevention Tip 6.1

A method that performs one task is easier to test and debug than one that performs many tasks.

Software Engineering Observation 6.3

If you cannot choose a concise name that expresses a method's task, your method might be attempting to perform too many tasks. Break such a method into several smaller methods.

6.2 Program Modules in Java (Cont.)

- Hierarchical form of management (Fig. 6.1).
  - A boss (the caller) asks a worker (the called method) to perform a task and report back (return) the results after completing the task.
  - The boss method does not know how the worker method performs its designated tasks.
  - The worker may also call other worker methods, unknown to the boss.
- “Hiding” of implementation details promotes good software engineering.

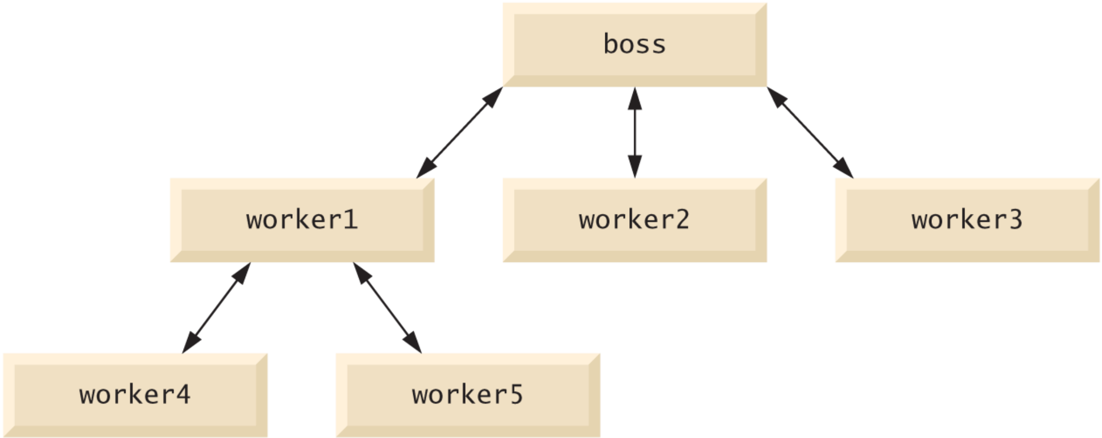

Fig. 6.1 | Hierarchical boss-method/worker-method relationship.

6.3 `static` Methods, `static` Fields and Class `Math`

- Sometimes a method performs a task that does not depend on the contents of any object.
  - Applies to the class in which it's declared as a whole
  - Known as a `static` method or a class method
- It's common for classes to contain convenient `static` methods to perform common tasks.
- To declare a method as `static`, place the keyword `static` before the return type in the method's declaration.
- Calling a `static` method
  - *`ClassName.methodName( arguments )`*
- Class `Math` provides a collection of `static` methods that enable you to perform common mathematical calculations.
- Method arguments may be constants, variables or expressions.

Method	Description	Example

`abs( x )`	absolute value of *x*	`abs( 23.7 )` is `23.7`
		`abs( 0.0 )` is `0.0`
		`abs( -23.7 )` is `23.7`

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of *x* (base *e*)	`log( Math.E )` is `1.0` `log( Math.E * Math.E )` is `2.0`
`max( x, y )`	larger value of *x* and *y*	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of *x* and *y*	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`

Fig. 6.2 | `Math` class methods. (Part 1 of 2.)

Method	Description	Example
`pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, 0.5 )` is `3.0`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 6.2 | `Math` class methods. (Part 2 of 2.)

6.4 static Methods, static Fields and Class Math (Cont.)

- Math fields for common mathematical constants
  - Math.PI (3.141592653589793)
  - Math.E (2.718281828459045)
- Declared in class Math with the modifiers public, final and static
  - public allows you to use these fields in your own classes.
  - A field declared with keyword final is constant—its value cannot change after the field is initialized.
  - PI and E are declared final because their values never change.

6.4 `static` Methods, `static` Fields and Class `Math` (Cont.)

- A field that represents an attribute is also known as an instance variable — each object (instance) of the class has a separate instance of the variable in memory.
- Fields for which each object of a class does not have a separate instance of the field are declared `static` and are also known as class variables.
- All objects of a class containing `static` fields share one copy of those fields.
- Together the class variables (i.e., `static` variables) and instance variables represent the fields of a class.

6.4 `static` Methods, `static` Fields and Class `Math` (Cont.)

- Why is method **`main`** declared **`static`**?
  - The JVM attempts to invoke the `main` method of the class you specify — when no objects of the class have been created.
  - Declaring `main` as `static` allows the JVM to invoke `main` without creating an instance of the class.

6.5 Declaring Methods with Multiple Parameters

- Multiple parameters are specified as a comma-separated list.
- There must be one argument in the method call for each parameter (sometimes called a formal parameter) in the method declaration.
- Each argument must be consistent with the type of the corresponding parameter.

```java
// Fig. 6.3: MaximumFinder.java
// Programmer-declared method maximum with three double parameters.
import java.util.Scanner;

public class MaximumFinder
{
   // obtain three floating-point values and locate the maximum value
   public void determineMaximum()
   {
      // create Scanner for input from command window
      Scanner input = new Scanner( System.in );

      // prompt for and input three floating-point values
      System.out.print(
         "Enter three floating-point values separated by spaces: " );
      double number1 = input.nextDouble(); // read first double
      double number2 = input.nextDouble(); // read second double
      double number3 = input.nextDouble(); // read third double

      // determine the maximum value
      double result = maximum( number1, number2, number3 );

```

Passing three arguments to method `maximum`

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 1 of 2.)

```java
        // display maximum value
        System.out.println( "Maximum is: " + result );
    } // end method determineMaximum

    // returns the maximum of its three double parameters
    public double maximum( double x, double y, double z )
    {
        double maximumValue = x; // assume x is the largest to start

        // determine whether y is greater than maximumValue
        if ( y > maximumValue )
            maximumValue = y;

        // determine whether z is greater than maximumValue
        if ( z > maximumValue )
            maximumValue = z;

        return maximumValue;
    } // end method maximum
} // end class MaximumFinder
```

Method `maximum` receives three parameters and returns the largest of the three

Fig. 6.3 | Programmer-declared method `maximum` with three `double` parameters. (Part 2 of 2.)

```java
// Fig. 6.4: MaximumFinderTest.java
// Application to test class MaximumFinder.

public class MaximumFinderTest
{
   // application starting point
   public static void main( String[] args )
   {
      MaximumFinder maximumFinder = new MaximumFinder();
      maximumFinder.determineMaximum();
   } // end main
} // end class MaximumFinderTest
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Fig. 6.4 | Application to test class `MaximumFinder`.

Software Engineering Observation 6.5

Methods can return at most one value, but the returned value could be a reference to an object that contains many values.

Software Engineering Observation 6.6

Variables should be declared as fields of a class only if they are required for use in more than one method of the class or if the program should save their values between calls to the class's methods.

Common Programming Error 6.1

Declaring method parameters of the same type as `float x, y` *instead of* `float x, float y` *is a syntax error—a type is required for each parameter in the parameter list.*

6.5 Declaring Methods with Multiple Parameters (Cont.)

- Implementing method `maximum` by reusing method `Math.max`
  - Two calls to `Math.max`, as follows:
    - `return Math.max( x, Math.max( y, z ) );`
  - The first specifies arguments `x` and `Math.max( y, z )`.
  - Before any method can be called, its arguments must be evaluated to determine their values.
  - If an argument is a method call, the method call must be performed to determine its return value.
  - The result of the first call is passed as the second argument to the other call, which returns the larger of its two arguments.

6.5 Declaring Methods with Multiple Parameters (Cont.)

- String concatenation
  - Assemble `String` objects into larger strings with operators `+` or `+=`.
- When both operands of operator `+` are `String`s, operator `+` creates a new `String` object
  - characters of the right operand are placed at the end of those in the left operand
- Every primitive value and object in Java has a `String` representation.
- When one of the `+` operator's operands is a `String`, the other is converted to a `String`, then the two are concatenated.
- If a `boolean` is concatenated with a `String`, the `boolean` is converted to the `String` `"true"` or `"false"`.
- All objects have a `toString` method that returns a `String` representation of the object.

6.6 Notes on Declaring and Using Methods (Cont.)

- A non-`static` method can call any method of the same class directly and can manipulate any of the class's fields directly.

- A `static` method can call *only other* ***`static`*** *methods* of the same class directly and can manipulate *only* ***`static`*** *fields* in the same class directly.
  - To access the class's non-`static` members, a `static` method must use a reference to an object of the class.

Common Programming Error 6.4

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error 6.5

Omitting the return-value-type, *possibly* `void`, *in a method declaration is a syntax error.*

Common Programming Error 6.6

Placing a semicolon after the right parenthesis enclosing the parameter list of a method declaration is a syntax error.

Common Programming Error 6.7

Redeclaring a parameter as a local variable in the method's body is a compilation error.

Common Programming Error 6.8

Forgetting to return a value from a method that should return a value is a compilation error. If a return type other than `void` *is specified, the method must contain a* `return` *statement that returns a value consistent with the method's return type. Returning a value from a method whose return type has been declared* `void` *is a compilation error.*

6.8 Argument Promotion and Casting

- Argument promotion
  - Converting an argument's value, if possible, to the type that the method expects to receive in its corresponding parameter.
- Conversions may lead to compilation errors if Java's promotion rules are not satisfied.
- Promotion rules
  - specify which conversions are allowed.
  - apply to expressions containing values of two or more primitive types and to primitive-type values passed as arguments to methods.
- Each value is promoted to the "highest" type in the expression.
- Figure 6.5 lists the primitive types and the types to which each can be promoted.

Type	Valid promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	int, long, float or double (but not char)
byte	short, int, long, float or double (but not char)
boolean	None (boolean values are not considered to be numbers in Java)

Fig. 6.5 | Promotions allowed for primitive types.

6.8 Argument Promotion and Casting (Cont.)

- Converting values to types lower in the table of Fig. 6.5 will result in different values if the lower type cannot represent the value of the higher type

- In cases where information may be lost due to conversion, the Java compiler requires you to use a cast operator to explicitly force the conversion to occur — otherwise a compilation error occurs.

Common Programming Error 6.9

Converting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, converting a floating-point value to an integer value may introduce truncation errors (loss of the fractional part) into the result.

6.9 Java API Packages

- Java contains many predefined classes that are grouped into categories of related classes called packages.
- A great strength of Java is the Java API's thousands of classes.
- Some key Java API packages are described in Fig. 6.6.
- Overview of the packages in Java SE 8:
  - `java.sun.com/javase/8/docs/api/overview-summary.html`
- Java API documentation
  - `java.sun.com/javase/8/docs/api/`

Package	Description
`java.applet`	The Java Applet Package contains a class and several interfaces required to create Java applets—programs that execute in web browsers. Applets are discussed in Chapter 23, Applets and Java Web Start; interfaces are discussed in Chapter 10, Object-Oriented Programming: Polymorphism.)
`java.awt`	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs in early versions of Java. In current versions of Java, the Swing GUI components of the `javax.swing` packages are typically used instead. (Some elements of the `java.awt` package are discussed in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D™, and Chapter 25, GUI Components: Part 2.)
`java.awt.event`	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (See Chapter 14, GUI Components: Part 1 and Chapter 25, GUI Components: Part 2.)

Fig. 6.6 | Java API packages (a subset). (Part 1 of 4.)

Package	Description
`java.awt.geom`	The **Java 2D Shapes Package** contains classes and interfaces for working with Java's advanced two-dimensional graphics capabilities. (See Chapter 15, Graphics and Java 2D™.)
`java.io`	The **Java Input/Output Package** contains classes and interfaces that enable programs to input and output data. (See Chapter 17, Files, Streams and Object Serialization.)
`java.lang`	The **Java Language Package** contains classes and interfaces (discussed bookwide) that are required by many Java programs. This package is imported by the compiler into all programs.
`java.net`	The **Java Networking Package** contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (See Chapter 27, Networking.)
`java.sql`	The **JDBC Package** contains classes and interfaces for working with databases. (See Chapter 28, Accessing Databases with JDBC.)

Fig. 6.6 | Java API packages (a subset). (Part 2 of 4.)

Package	Description
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to locales (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`) and the storing and processing of large amounts of data. (See Chapter 20, Generic Collections.)
`java.util.concurrent`	The Java Concurrency Package contains utility classes and interfaces for implementing programs that can perform multiple tasks in parallel. (See Chapter 26, Multithreading.)
`javax.media`	The Java Media Framework Package contains classes and interfaces for working with Java's multimedia capabilities. (See Chapter 24, Multimedia: Applets and Applications.)

Fig. 6.6 | Java API packages (a subset). (Part 3 of 4.)

Package	Description
`javax.swing`	The **Java Swing GUI Components Package** contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (See Chapter 14, GUI Components: Part 1 and Chapter 25, GUI Components: Part 2.)
`javax.swing.event`	The **Java Swing Event Package** contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (See Chapter 14, GUI Components: Part 1 and Chapter 25, GUI Components: Part 2.)
`javax.xml.ws`	The **JAX-WS Package** contains classes and interfaces for working with web services in Java. (See Chapter 31, Web Services.)

Fig. 6.6 | Java API packages (a subset). (Part 4 of 4.)

6.10 Case Study: Random-Number Generation

- Simulation and game playing
  - element of chance
  - Class `Random` (package `java.util`)
  - `static` method `random` of class `Math`.
- Objects of class `Random` can produce random `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values
- `Math` method `random` can produce only `double` values in the range $0.0 \le x < 1.0$.
- Documentation for class `Random`
  - `java.sun.com/javase/6/docs/api/java/util/Random.html`

6.10 Case Study: Random-Number Generation (Cont.)

- Class `Random` produces pseudorandom numbers
  - A sequence of values produced by a complex mathematical calculation.
  - The calculation uses the current time of day to seed the random-number generator.
- The range of values produced directly by `Random` method `nextInt` often differs from the range of values required in a particular Java application.
- `Random` method `nextInt` that receives an `int` argument returns a value from 0 up to, but not including, the argument's value.

6.10 Case Study: Random-Number Generation (Cont.)

- Rolling a Six-Sided Die
  - `face = 1 + randomNumbers.nextInt( 6 );`
  - The argument `6` — called the scaling factor — represents the number of unique values that `nextInt` should produce (0–5)
  - This is called scaling the range of values
  - A six-sided die has the numbers 1–6 on its faces, not 0–5.
  - We shift the range of numbers produced by adding a shifting value—in this case 1—to our previous result, as in
  - The shifting value (`1`) specifies the first value in the desired range of random integers.

```java
// Fig. 6.7: RandomIntegers.java
// Shifted and scaled random integers.
import java.util.Random; // program uses class Random

public class RandomIntegers
{
   public static void main( String[] args )
   {
      Random randomNumbers = new Random(); // random number generator
      int face; // stores each random integer generated

      // loop 20 times
      for ( int counter = 1; counter <= 20; counter++ )
      {
         // pick random integer from 1 to 6
         face = 1 + randomNumbers.nextInt( 6 );

         System.out.printf( "%d  ", face ); // display generated value

         // if counter is divisible by 5, start a new line of output
         if ( counter % 5 == 0 )
            System.out.println();
      } // end for
   } // end main
} // end class RandomIntegers
```

Program uses class `Random` from package `java.util`

Creates a `Random` object

Produces integers in the range 1 through 6

Fig. 6.7 | Shifted and scaled random integers. (Part 1 of 2.)

```
1    5    3    6    2
5    2    6    5    2
4    4    4    2    6
3    1    6    2    2
```

```
6    5    4    2    6
1    2    5    1    3
6    3    2    2    1
6    4    2    6    4
```

Fig. 6.7 | Shifted and scaled random integers. (Part 2 of 2.)

6.10 Case Study: Random-Number Generation (Cont.)

- Fig 6.8: Rolling a Six-Sided Die 6000 Times

```java
// Fig. 6.8: RollDie.java
// Roll a six-sided die 6000 times.
import java.util.Random;

public class RollDie
{
   public static void main( String[] args )
   {
      Random randomNumbers = new Random(); // random number generator

      int frequency1 = 0; // maintains count of 1s rolled
      int frequency2 = 0; // count of 2s rolled
      int frequency3 = 0; // count of 3s rolled
      int frequency4 = 0; // count of 4s rolled
      int frequency5 = 0; // count of 5s rolled
      int frequency6 = 0; // count of 6s rolled

      int face; // stores most recently rolled value

      // tally counts for 6000 rolls of a die
      for ( int roll = 1; roll <= 6000; roll++ )
      {
         face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6

```

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 1 of 3.)

```
      // determine roll value 1-6 and increment appropriate counter
      switch ( face )
      {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break; // optional at end of switch
      } // end switch
   } // end for

```

Value from 1 through 6 used to update appropriate counter

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 2 of 3.)

```
         System.out.println( "Face\tFrequency" ); // output headers
         System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
            frequency1, frequency2, frequency3, frequency4,
            frequency5, frequency6 );
      } // end main
   } // end class RollDie
```

```
Face     Frequency
1        982
2        1001
3        1015
4        1005
5        1009
6        988
```

```
Face     Frequency
1        1029
2        994
3        1017
4        1007
5        972
6        981
```

Fig. 6.8 | Rolling a six-sided die 6000 times. (Part 3 of 3.)

6.11 Case Study: A Game of Chance; Introducing Enumerations

- Basic rules for the dice game Craps:
  - *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.*

```java
// Fig. 6.9: Craps.java
// Craps class simulates the dice game craps.
import java.util.Random;

public class Craps
{
   // create random number generator for use in method rollDice
   private static final Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST };

   // constants that represent common rolls of the dice
   private static final int SNAKE_EYES = 2;
   private static final int TREY = 3;
   private static final int SEVEN = 7;
   private static final int YO_LEVEN = 11;
   private static final int BOX_CARS = 12;

```

Declares constants for the game status

Declares constants representing common rolls of the dice

Fig. 6.9 | Craps class simulates the dice game craps. (Part 1 of 4.)

```java
    // plays one game of craps
    public void play()
    {
        int myPoint = 0; // point if no win or loss on first roll
        Status gameStatus; // can contain CONTINUE, WON or LOST

        int sumOfDice = rollDice(); // first roll of the dice

        // determine game status and point based on first roll
        switch ( sumOfDice )
        {
            case SEVEN: // win with 7 on first roll
            case YO_LEVEN: // win with 11 on first roll
                gameStatus = Status.WON;
                break;
            case SNAKE_EYES: // lose with 2 on first roll
            case TREY: // lose with 3 on first roll
            case BOX_CARS: // lose with 12 on first roll
                gameStatus = Status.LOST;
                break;
```

Variable that stores the game status

Roll the dice to start playing the game

Player wins on the first roll; set `gameStatus` to `WON`

Player loses on the first roll; set `gameStatus` to `LOST`

Fig. 6.9 | `Craps` class simulates the dice game craps. (Part 2 of 4.)

```java
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            System.out.printf( "Point is %d\n", myPoint );
            break; // optional at end of switch
      } // end switch

      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
      {
         sumOfDice = rollDice(); // roll dice again

         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
               gameStatus = Status.LOST;
      } // end while

```

Player did not win or lose; set `gameStatus` to `CONTINUE`

Loop while game is not over

Roll the dice again

Made your point; set `gameStatus` to `WON`

Rolled 7; set `gameStatus` to `WON`

Fig. 6.9 | `Craps` class simulates the dice game craps. (Part 3 of 4.)

```java
        // display won or lost message
        if ( gameStatus == Status.WON )
            System.out.println( "Player wins" );
        else
            System.out.println( "Player loses" );
    } // end method play

    // roll dice, calculate sum and display results
    public int rollDice()
    {
        // pick random die values
        int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
        int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll

        int sum = die1 + die2; // sum of die values

        // display results of this roll
        System.out.printf( "Player rolled %d + %d = %d\n",
            die1, die2, sum );

        return sum; // return sum of dice
    } // end method rollDice
} // end class Craps
```

Display a message indicating whether the user won or lost

Fig. 6.9 | Craps class simulates the dice game craps. (Part 4 of 4.)

```java
// Fig. 6.10: CrapsTest.java
// Application to test class Craps.

public class CrapsTest
{
   public static void main( String[] args )
   {
      Craps game = new Craps();
      game.play(); // play one game of craps
   } // end main
} // end class CrapsTest
```

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

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

Fig. 6.10 | Application to test class `Craps`. (Part 1 of 2.)

```
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. 6.10 | Application to test class `Craps`. (Part 2 of 2.)

6.11 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

- Notes:
  - `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 is assigned a value.
  - `gameStatus` is assigned a value in every `case` of the `switch` statement—thus, it's guaranteed to be initialized before it's used and does not need to be initialized.

6.11 Case Study: A Game of Chance; Introducing Enumerations (Cont.)

- `enum` type `Status`
  - An enumeration in its simplest form declares a set of constants represented by identifiers.
  - Special kind of class that is introduced by the keyword `enum` and a type name.
  - Braces delimit an `enum` declaration's body.
  - Inside the braces is a comma-separated list of enumeration constants, each representing a unique value.
  - The identifiers in an `enum` must be unique.
  - Variables of an `enum` type can be assigned only the constants declared in the enumeration.

Good Programming Practice 6.1

Use only uppercase letters in the names of enumeration constants. This makes the constants stand out and reminds you that enumeration constants are not variables.

Good Programming Practice 6.2

Using enumeration 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.*

6.12 Scope of Declarations

- Declarations introduce names that can be used to refer to such Java entities.
- The scope of a declaration is the portion of the program that can refer to the declared entity by its name.
  - Such an entity is said to be “in scope” for that portion of the program.

6.12 Scope of Declarations (Cont.)

- Basic scope rules:
  - The scope of a parameter declaration is the body of the method in which the declaration appears.
  - The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
  - The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.
  - A method or field's scope is the entire body of the class.
- Any block may contain variable declarations.
- If a local variable or parameter in a method has the same name as a field of the class, the field is "hidden" until the block terminates execution—this is called shadowing.

Common Programming Error 6.10

A compilation error occurs when a local variable is declared more than once in a method.

Error-Prevention Tip 6.3

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field in the class.

```java
// Fig. 6.11: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private int x = 1;

   // method begin creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public void begin()
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf( "local x in method begin is %d\n", x );

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf( "\nlocal x in method begin is %d\n", x );
   } // end method begin
```

Class scope

Method scope

Fig. 6.11 | Scope class demonstrating scopes of a field and local variables. (Part 1 of 2.)

```java

   // create and initialize local variable x during each call
   public void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

      System.out.printf(
         "\nlocal x on entering method useLocalVariable is %d\n", x );
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d\n", x );
   } // end method useLocalVariable

   // modify class Scope's field x during each call
   public void useField()
   {
      System.out.printf(
         "\nfield x on entering method useField is %d\n", x );
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d\n", x );
   } // end method useField
} // end class Scope
```

Method scope

Uses instance variable x

Fig. 6.11 | Scope class demonstrating scopes of a field and local variables. (Part 2 of 2.)

```java
Fig. 6.12: ScopeTest.java
// Application to test class Scope.

public class ScopeTest
{
   // application starting point
   public static void main( String[] args )
   {
      Scope testScope = new Scope();
      testScope.begin();
   } // end main
} // end class ScopeTest
```

Fig. 6.12 | Application to test class Scope. (Part 1 of 2.)

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

Fig. 6.12 | Application to test class Scope. (Part 2 of 2.)

6.13 Method Overloading

- Method overloading
  - Methods of the same name declared in the same class
  - Must have different sets of parameters
- Compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call.
- Used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.
- Literal integer values are treated as type `int`, so the method call in line 9 invokes the version of `square` that specifies an `int` parameter.
- Literal floating-point values are treated as type `double`, so the method call in line 10 invokes the version of `square` that specifies a `double` parameter.

```java
// Fig. 6.13: MethodOverload.java
// Overloaded method declarations.

public class MethodOverload
{
   // test overloaded square methods
   public void testOverloadedMethods()
   {
      System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
      System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
   } // end method testOverloadedMethods

   // square method with int argument
   public int square( int intValue )
   {
      System.out.printf( "\nCalled square with int argument: %d\n",
         intValue );
      return intValue * intValue;
   } // end method square with int argument

```

Calls `square` with an `int` parameter

Calls `square` with a `double` parameter

`square` method that receives an `int`

Fig. 6.13 | Overloaded method declarations. (Part 1 of 2.)

```java
    // square method with double argument
    public double square( double doubleValue )
    {
       System.out.printf( "\nCalled square with double argument: %f\n",
          doubleValue );
       return doubleValue * doubleValue;
    } // end method square with double argument
} // end class MethodOverload
```

square method that receives a double

Fig. 6.13 | Overloaded method declarations. (Part 2 of 2.)

```java
// Fig. 6.14: MethodOverloadTest.java
// Application to test class MethodOverload.

public class MethodOverloadTest
{
   public static void main( String[] args )
   {
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.testOverloadedMethods();
   } // end main
} // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Fig. 6.14 | Application to test class `MethodOverload`.

6.14 Method Overloading

- Distinguishing Between Overloaded Methods
  - The compiler distinguishes overloaded methods by their signatures — the methods' names and the number, types and order of their parameters.
- Return types of overloaded methods
  - *Method calls cannot be distinguished by return type.*
- Figure 6.15 illustrates the errors generated when two methods have the same signature and different return types.
- Overloaded methods can have different return types if the methods have different parameter lists.
- Overloaded methods need not have the same number of parameters.

```java
// Fig. 6.15: MethodOverloadError.java
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.

public class MethodOverloadError
{
   // declaration of method square with int argument
   public int square( int x )
   {
      return x * x;
   }

   // second declaration of method square with int argument
   // causes compilation error even though return types are different
   public double square( int y )
   {
      return y * y;
   }
} // end class MethodOverloadError
```

Generates a compilation error

Fig. 6.15 | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 1 of 2.)

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Fig. 6.15 | Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different. (Part 2 of 2.)

End of Part I