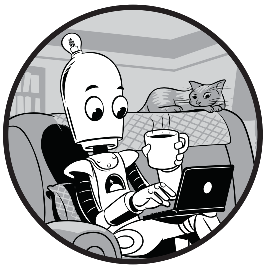
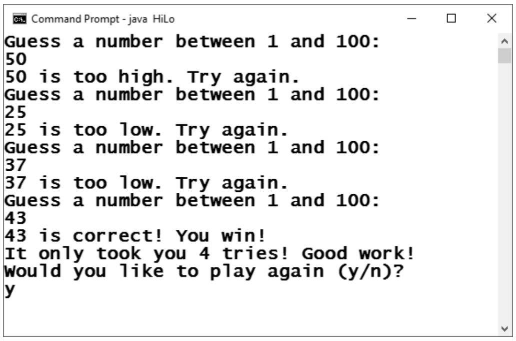
# 2

# BUILD A HI-LO GUESSING GAME APP!



Let's begin by coding a fun, playable game in Java: the Hi-Lo guessing game. We'll program this game as a *command line application*, which is just a fancy way of saying it's text based (see Figure 2-1). When the program runs, the prompt will ask the user to guess a number between 1 and 100. Each time they guess, the program will tell them whether the guess is too high, too low, or correct.



Command Prompt - java HiLo

```
Guess a number between 1 and 100:
50
50 is too high. Try again.
Guess a number between 1 and 100:
25
25 is too low. Try again.
Guess a number between 1 and 100:
37
37 is too low. Try again.
Guess a number between 1 and 100:
43
43 is correct! You win!
It only took you 4 tries! Good work!
Would you like to play again (y/n)?
y
```

*Figure 2-1: A text-based Hi-Lo guessing game*

Now that you know how the game works, all you have to do is code the steps to play it. We'll start by mapping out the app at a high level and then code a very simple version of the game. By starting out with a goal in mind and understanding how to play the game, you'll be able to pick up coding skills more easily, and you'll learn them with a purpose. You can also enjoy the game immediately after you finish coding it.

## Planning the Game Step-by-Step

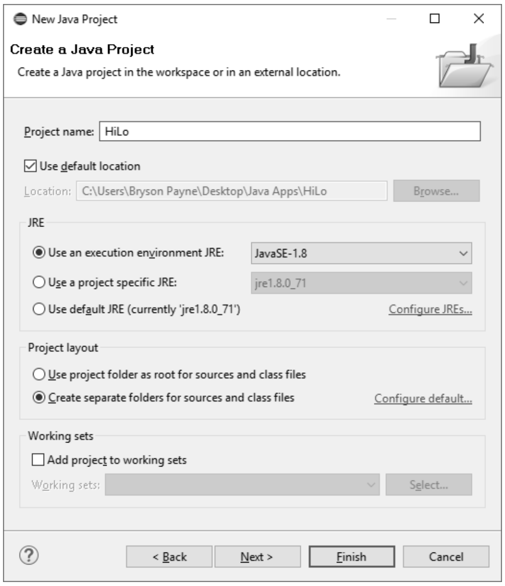
Let's think about all the steps we'll need to code in order to get the Hi-Lo guessing game to work. A basic version of the game will need to do the following:

1. Generate a random number between 1 and 100 for the user to guess.
2. Display a *prompt*, or a line of text, asking the user to guess a number in that range.
3. Accept the user's guess as input.
4. Compare the user's guess to the computer's number to see if the guess is too high, too low, or correct.
5. Display the results on the screen.
6. Prompt the user to guess another number until they guess correctly.
7. Ask the user if they'd like to play again.

We'll start with this basic structure. In Programming Challenge #2, you'll try adding an extra feature, to tell the user how many tries it took to guess the number correctly.

## Creating a New Java Project

The first step in coding a new Java app in Eclipse is creating a project. On the menu bar in Eclipse, go to **File ▸ New ▸ Java Project** (or select **File ▸ New ▸ Project**, then **Java ▸ Java Project** in the New Project wizard). The New Java Project dialog should pop up, as shown in Figure 2-2.


New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: HiLo

☑ Use default location

Location: C:\Users\Bryson Payne\Desktop\Java Apps\HiLo

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jre1.8.0_71

○ Use default JRE (currently 'jre1.8.0_71')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Working sets

☐ Add project to working sets

Working sets:

*Figure 2-2: The New Java Project dialog for the Hi-Lo guessing game app*

Type **`HiLo`** into the Project name field. Note that uppercase and lowercase letters are important in Java, and we'll get in the habit of using uppercase letters to start all of our project, file, and class names, which is a common Java practice. Leave all the other settings unchanged and click **Finish**. Depending on your version of Eclipse, you may be asked if you want to open the project using the Java Perspective. A *perspective* in Eclipse is a workspace set up for coding in a specific language. Click **Yes** to tell Eclipse you'd like the workspace set up for convenient coding in Java.

## Creating the HiLo Class

Java is an *object-oriented programming language*. Object-oriented programming languages use *classes* to design reusable pieces of programming code. Classes are like templates that make it easier to create *objects*, or instances of that class. If you think of a class as a cookie cutter, objects are the cookies. And, just like a cookie cutter, classes are reusable, so once we've built a useful class, we can reuse it over and over to create as many objects as we want.

The Hi-Lo guessing game will have a single class file that creates a guessing game object with all the code needed to play the game. We'll call our new class `HiLo`. The capitalization matters, and naming the class `HiLo` follows several Java naming conventions. It's common practice to start all class names with an uppercase letter, so we use a capital `H` in `HiLo`. Also, there should be no spaces, hyphens, or special characters between words in a class name. Finally, we use camel case for class names with multiple words, beginning each new word with a capital letter, as in `HiLo`, `GuessingGame`, and `BubbleDrawApp`. The words look like they have humps in the middle, just like a camel.

To create the new `HiLo` class, first find your *HiLo* project folder under the Package Explorer pane on the left side of the Eclipse workspace. Expand the folder by clicking the small arrow to the left of it. You should see a subfolder called *src*, short for *source code*. All the text files containing your Java programs will go in this *src* folder.

Right-click the *src* folder and select **New ▸ Class**, as shown in Figure 2-3.

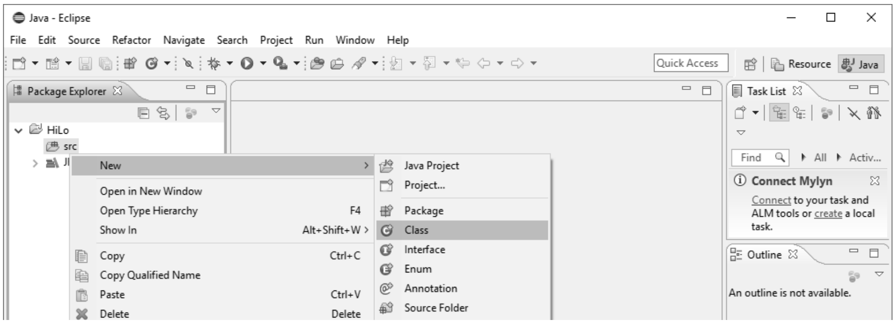
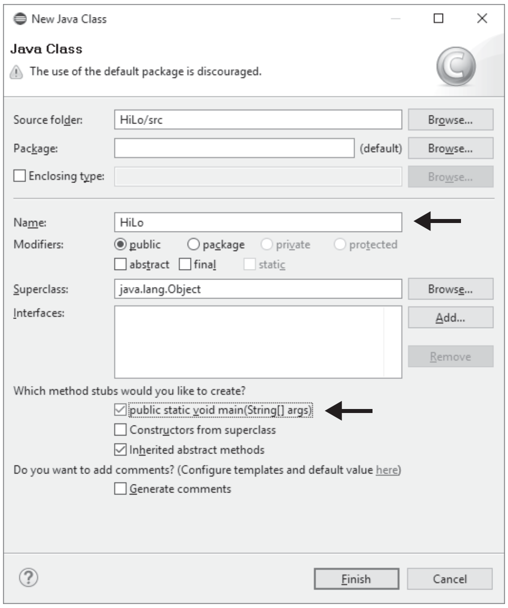


*Figure 2-3: Creating a new class file for the Hi-Lo guessing game app*

The New Java Class dialog will appear, as shown in Figure 2-4. Type **`HiLo`** into the Name field. Then, under *Which method stubs would you like to create?*, check the box for **public static void main(String[] args)**. This tells Eclipse that we're planning to write a `main()` program method, so Eclipse will include a *stub*, or skeleton, for the `main()` method that we can fill in with our own code. The `main()` method is required any time you want to run an app as a stand-alone program.



New Java Class

Java Class

The use of the default package is discouraged.

Source folder: HiLo/src

Package: (default)

Enclosing type:

Name: HiLo

Modifiers: [✓ public] package private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

[✓] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

*Figure 2-4: Name the new Java class* HiLo *and select the checkbox to create a* `main()` *method.*

Click **Finish** in the New Java Class dialog, and you should see a new file named *HiLo.java* that contains the code shown in Listing 2-1. This Java file will be the outline of the Hi-Lo guessing game. We'll write the guessing game program by editing this file and adding code inside it.

```
❶ public class HiLo {
    ❷ public static void main(String[] args) {
          // TODO Auto-generated method stub

      }
  }
```

*Listing 2-1: The stub code for the `HiLo` guessing game class, generated by Eclipse*

**NOTE** *The numbered circles point out important lines, but they aren't actually part of the code.*

Eclipse creates this code all on its own. The class `HiLo` is `public` ❶, meaning we can run it from the command line or terminal.

Java groups statements with braces, { and }. The opening brace, {, begins a block of statements that will form the body of the `HiLo` class. The closing brace, }, ends the block of statements. Inside the class is the `main()` method ❷, which is the method that will run when the class is executed.

Inside the opening brace for the `main()` method is a comment line that starts with `//`. Comments are for us (the humans) to read. They're ignored by the computer, so we can use them to help us remember what a section of code does or to leave notes for future use. You can delete the `TODO` comment in Listing 2-1.

## Generating a Random Number

The first programming task for our game is to generate a random number. We'll use the `Math` class, which contains a method for generating a random floating-point (decimal) number between 0.0 and 1.0. Then, we'll convert that decimal value to an *integer* (a whole number) between 1 and 100. The `Math` class is a built-in class that contains many useful math functions like the ones you might find on a nice scientific calculator.

Inside the `main()` method, add the comment and line of code shown in Listing 2-2.

```
public class HiLo {
    public static void main(String[] args) {
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
    }
}
```

*Listing 2-2: The code to create a random number between 1 and 100*

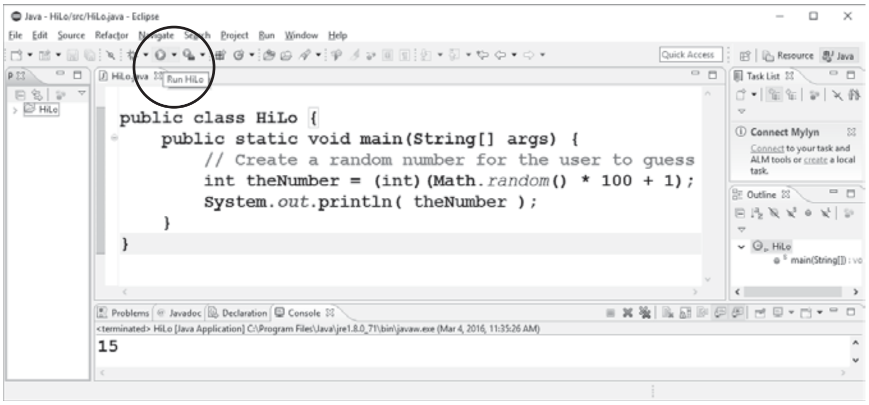
First, we need to create a variable to hold the random number the user will be trying to guess in the app. Since the app will ask the user to guess a whole number between 1 and 100, we'll use the `int` type, short for *integer*. We name our variable `theNumber`. The equal sign, `=`, assigns a value to our new `theNumber` variable. We use the built-in `Math.random()` function to generate a random number between 0.0 and just under 1.0 (0.99999). Because `Math.random()` generates numbers only in that specific range, we need to multiply the random number we get by 100 to stretch the range from 0.0 to just under 100.0 (99.99999 or so). Then we add 1 to that value to ensure the number runs from 1.0 (0.0 + 1) to 100.99999.

The `(int)` part is called a *type cast*, or just *cast* for short. Casting changes the *type* of the number from a decimal number to an integer. In this case, everything after the decimal point is removed, resulting in a whole number between 1 and 100. Java then stores that number in the variable `theNumber`, the number the user is trying to guess in the game. Finally, we add a semicolon (`;`) to indicate the end of the instruction.

Now, you can add a `System.out.println()` statement to print the number you've generated:

```
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
    }
}
```

After we add this line of code, we can run the program to see it generate and print a random number. Click the green run button in the top menu bar to compile and run the program, as shown in Figure 2-5. You can also go to the **Run** menu and select **Run**.


Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
HiLo.java
Run HiLo
HiLo
Task List
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
HiLo
main(String[]) : vo
Problems
Javadoc
Declaration
Console

```
public class HiLo {
    public static void main(String[] args) {
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        System.out.println( theNumber );
    }
}
```

```
<terminated> HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 4, 2016, 11:35:26 AM)
15
```

*Figure 2-5: Printing a random number to the screen*

Your random number will appear in the small console window at the bottom of the screen, as shown in Figure 2-5. If you run your program again, you'll see a different number between 1 and 100.

This would be a great time to play with the program a bit. Try generating a number between 1 and 10, or 1 and 1,000—even 1 to 1,000,000. Java will accept numbers all the way to a billion or so. Just remember to write your numbers without commas: 1,000 becomes `1000` in Java, and 1,000,000 is written `1000000`. You probably don't want to guess a number between 1 and 1,000,000 the first time you play the game, though, so remember to change this line back before you move ahead.

**NOTE** *Remember to save your code often. Eclipse will save for you automatically every time you run a program, but it's a good idea to save after every few lines of code. In fact, pressing* CTRL-S *to save after each line of code isn't a bad habit to get into. I've never heard a coder say they wish they hadn't saved so often, but I've experienced losing unsaved code a few times myself, and it's not fun. Save often, and remember that you can use* ***Edit*** ▸ ***Undo*** *if you ever type something incorrectly or accidentally delete a section of code.*

## Getting User Input from the Keyboard

Now let's add the code that allows the user to guess a number. To do this, we'll need to *import* some additional Java capabilities. Java comes with many libraries and packages that we can use in our own projects. Libraries and packages are sets of code that someone else has created. When we import them, we get new features that make creating our own programs even easier. We can access packages and libraries whenever we need them using the `import` statement.

For the guessing game program, we need to be able to accept keyboard input from the user. The `Scanner` class, contained in the `java.util` utilities package, provides several useful functions for working with keyboard input. Let's import the `Scanner` class into our program. Add the following statement at the top of the *HiLo.java* file, before the line `public class HiLo`:

```
import java.util.Scanner;

public class HiLo {
```

This line imports the `Scanner` class and all its functionality from the main Java utilities package. The `Scanner` class includes functions like `nextLine()` to accept a line of input from the keyboard and `nextInt()` to turn text input from the keyboard into an integer number that can be compared or used in calculations. To use the `Scanner` class for keyboard input, we have to tell it to use the keyboard as its source.

We want to do this before anything else in the program, so add this line of code inside the top of the `main()` method:

```
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
```

This line creates a `Scanner` object called `scan` that pulls input from the computer's keyboard, `System.in`.

Although this new line of code sets up the scan object, it doesn't actually ask for input yet. To get the user to type in a guess, we'll need to *prompt* them by asking them to enter a number. Then, we'll take the number they enter from the keyboard and store it in a variable that we can compare against theNumber, the computer's original random number. Let's call the variable that will store the user's guess something easy to remember, like guess. Add the following line next:

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    // Create a random number for the user to guess
    int theNumber = (int)(Math.random() * 100 + 1);
    System.out.println( theNumber );
    int guess = 0;
```

This statement both *declares* a variable called guess of type int (an integer in Java), and it *initializes* the guess variable to a starting value of 0. Some programming languages require a variable to be declared and then initialized in separate lines of code, but Java allows programmers to include both the declaration and initialization of variables in a single line. Java requires every variable to be declared with a specific *type*, or kind of information it should store. The user's guess will be a whole number, so we've declared guess as an int.

Next, we need to prompt the user to enter a guess. We can let the user know the program is ready for input by printing a line of text to the console window (or command line). We access this text-based screen as a part of our computer system through the System class, just like we did for keyboard input. But this time, we want to *output* information for the user to read. The object that lets us access the command line console for output is System.out. Similar to the System.in object that allows us to receive text input from the keyboard, System.out gives us the ability to output text to the screen. The specific function to print a line of text is the println() command:

```
    int guess = 0;
    System.out.println("Guess a number between 1 and 100:");
```

Here we are using *dot notation*, which lists a class or object, followed by a dot and then a method or an attribute of that class or object. *Methods* are the functions in an object or class. Methods need to be called with dot notation to tell Java which object or class they belong to. *Attributes* are the values stored in an object or class.

For example, System is a class representing your computer system. System.out is the command line screen object contained in the System class, because your computer monitor is part of your overall computer system. System.out.println() is a method to print a line of text using the System.out object. We'll get more practice using dot notation as we continue.

Now that the user knows what kind of input the program is expecting from them, it's time to check the keyboard for their guess. We'll use the Scanner object called scan that we created earlier. Scanners have a method called nextInt() that looks for the next int value the user inputs from the keyboard. We'll store the user's guess in the variable guess that we created earlier:

```
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
```

This statement will wait for the user to type something into the console window (hopefully a whole number between 1 and 100—we'll see how to make sure the user enters a valid number in the next chapter) and press ENTER. The nextInt() method will take the string of text characters the user typed ("50", for example), turn it into the correct numeric value (50), and then store that number in the variable guess. Let's take a moment to save the changes we've made so far by going to **File ▸ Save** or pressing CTRL-S.

## *Making the Program Print Output*

We can also check to make sure our program is working so far by adding another println() statement:

```
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
```

This line uses the System.out.println() method again, but now we're combining text and numeric output. If the user guesses 50, we want the output to read, "You entered 50." To make this happen, we form a println() statement that mixes text with the number stored in the variable guess.

Java allows us to concatenate strings of text using the plus-sign operator, +. We use double quotation marks to specify the text we want to output first ("You entered "). Note the space before the closing quotation marks—this tells the program that we want a space to appear in the printed output after the last word. Java ignores most spacing, but when a space is included inside the quotation marks of a string of text, it becomes part of that text.

We also want to print the number the user guessed. We've stored this value in the variable called guess, so we just have to use the println() statement to output that value. Fortunately, in Java, when you include a variable in a println() statement, Java prints the value contained in that variable. So, immediately after the text "You entered ", we add the concatenation operator + followed by the variable name guess. Finally, we want to end the sentence with a period, so we use another concatenation operator + followed by the text we want, contained in double quotation marks, so it looks like ".".

Listing 2-3 puts together all of our lines of code so far.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
      ❶ // System.out.println( theNumber );
        int guess = 0;
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
    }
}
```

*Listing 2-3: The code to this point generates a random number and allows the user to guess once.*

At ❶, note that I turned `System.out.println( theNumber );` into a comment by adding a pair of forward slashes to the start of that line. This is called *commenting out*, and it's a useful technique for *debugging*—finding and fixing bugs or errors in programs. We used this `println()` statement earlier to show the value of the variable `theNumber` while we were writing and testing the program. Now, rather than deleting the line entirely, we can turn it into a comment so it's ignored by the computer. If we want to use that line again, we can just remove the `//` to include it in the program.

Now let's save our program and run it to see how it works so far. To run it, press the green run button or go to **Run ▸ Run**. Right now, the user can guess only once and the program doesn't check whether they guessed correctly. So next, we'll add some code so that the user can guess more than one time, and then we'll learn how to test each guess against `theNumber`.

## Loops: Ask, Check, Repeat

To give the user more than one chance to guess the number, we need to learn how to build a loop! In the guessing game program, we need to ask the user for a guess until they guess correctly. *Loops* give us the ability to repeat a set of steps over and over. In this section, we'll build a loop for the steps that prompt the user for a guess and accept the keyboard input.

Loops are very powerful programming tools, and they're one of the reasons computers are so valuable in our daily lives and in the business world—computers are really good at repeating the same task predictably. And, if they're programmed correctly, they can do this all day, every day, without making mistakes. You or I might get tired of telling someone their guess is too high or too low, but the computer never does. It will also never forget the number or tell the player their guess is too low or too high when it's actually not.

Let's tap into the power of loops with a `while` loop. A `while` loop repeats a set of statements as long as some *condition* is true. A condition is just something we can test. For example, in this program, we want to know whether the user correctly guessed the secret number. If they didn't guess correctly, we want to keep giving them a chance to guess again until they get it right.

To write a `while` loop, we need to know what condition we want to test for before repeating the loop each time. In the guessing game, we want the user to guess again as long as their guess isn't equal to the secret number theNumber. When the user's guess is equal to the secret number, the user wins and the game is over, so the loop should stop.

To create a while loop, we need to insert a while statement before the last three lines of code and then wrap the three lines for guessing inside a new pair of braces, as follows:

```
        int guess = 0;
        while(guess != theNumber)
        {
        System.out.println("Guess a number between 1 and 100:");
        guess = scan.nextInt();
        System.out.println("You entered " + guess + ".");
        }
    }
}
```

We use the keyword while to let Java know we're building a while loop, and then we put the appropriate condition inside parentheses. The part inside the parentheses, guess != theNumber, means that while the value stored in guess is not equal to (!=) the value stored in theNumber, the loop should repeat whatever statement or set of statements immediately follow this line of code. The operator != is a *comparison* operator—in this case, it compares guess and theNumber and evaluates whether they're different, or *not equal*. You'll learn about other comparison operators in the next section, but this is the one we need for the guessing while loop.

We need to tell Java what statements to repeat in the while loop, so I've added an opening brace, {, on the line after the while statement. In the same way that braces group all the statements together in the main() method, these braces group statements together inside the while loop.

There are three statements that we want to include inside the loop. First we need the println() statement that prompts the user to guess a number. Then we need the statement that scans the keyboard and records the guess with the nextInt() method. Finally, we need the println() statement that tells the user what they entered. To turn this set of statements into a block of code that will be run repeatedly in the while statement, we write the while statement and condition first, then an opening brace, then all three statements, and finally, a closing brace. Don't forget the closing brace! Your program won't run if it's missing.

One good programming practice that will help you keep your code organized and readable is using tab spacing correctly. Highlight the three statements inside the braces for the while statement and then press the TAB key to indent them.

The result should look like the following code:

```
        int guess = 0;
        while(guess != theNumber)
        {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
```

```
                System.out.println("You entered " + guess + ".");
            }
        }
    }
```

Correct indentation will help you remember to match up your opening and closing braces, and it will help you quickly see which statements are inside a loop or other block of code, as well as which statements are outside the loop. Indentation doesn't affect how your program runs, but if done well, it makes your program much easier to read and maintain.

Save your program now and run it to check that it works. The game is almost playable now, but we still need to tell the program to check if the user's guess is too high, too low, or just right. Time for (drum roll, please . . .) `if` statements!

## *if Statements: Testing for the Right Conditions*

Now that the user is able to guess until they are correct, we need to check the guess to let them know whether they were too high or too low. The statement that allows us to do this is the `if` statement.

An `if` statement will select whether to run a block of statements once or not at all based on a condition, or a *conditional expression*.

We used a conditional expression before in the guessing loop: (`guess != theNumber`). To check whether a guess is too high or too low, we just need a few more comparison operators: less than (`<`), greater than (`>`), and equal to (`==`).

First, instead of just telling the user what their guess was, let's write some code to check whether their guess was too low. Replace the last line of the `while` statement with the following two-line `if` statement:

```
        while (guess != theNumber)
        {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
        }
```

The `if` statement begins with the keyword `if`, followed by a conditional expression in parentheses. In this case, the condition is `guess < theNumber`, which means the value of the user's guess is less than the value of the random secret number. Notice there's no semicolon after the parentheses, because the `println()` statement that follows is actually part of the `if` statement. The whole statement tells the program that if the condition is true, it should print the user's guess and let them know they guessed too low. We use the concatenation operator (`+`) between the user's guess and the string of text telling them the guess was too low. Note the space after the first double quote and before `is`. This separates the user's guess from the word `is`.

If you run the program now and enter a low guess, like **1**, the if statement should tell the program to say your guess is too low. That's a good start, but what if we guess a number that's too high instead? In that case, we need an else statement.

The else statement gives the program a way to choose an alternative path, or set of steps, if the condition in the if statement is not true. We can test for guesses that are too high or too low with an if-else statement pair. Let's add an else statement right after the if statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
❷ else if (guess > theNumber)
      System.out.println(guess + " is too high. Try again.");
```

Notice that the code at ❷ looks similar to the code at ❶. Often when we're using if-else statements, we need to check for multiple conditions in a row, instead of just one. Here, we need to check for a guess that's too low, too high, or just right. In cases like this, we can chain if-else conditions together by placing the next if statement inside the else portion of the previous if-else statement. At ❷ we've begun the next if statement immediately after the else from the previous condition. If the guess is higher than the number, the program tells the user their guess is too high. Now that the program can tell the user if their guess is too high or too low, we just need to tell them if they guessed correctly and won!

If neither of the previous conditions is true—the user's guess is not too high and not too low—then they must have guessed the number. So we add one final else statement:

```
❶ if (guess < theNumber)
      System.out.println(guess + " is too low. Try again.");
❷ else if (guess > theNumber)
      System.out.println(guess + " is too high. Try again.");
❸ else
      System.out.println(guess + " is correct. You win!");
```

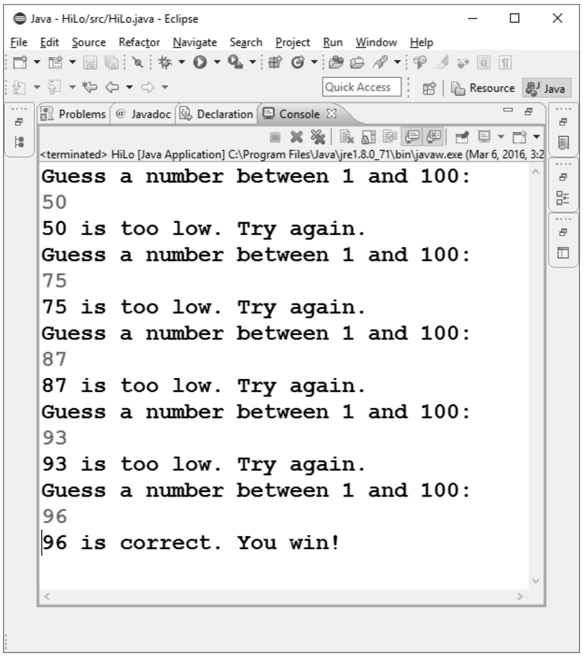
Notice that we don't need a conditional expression for this final else statement ❸. A correct guess is the only remaining option if the number is neither too high nor too low. In the case of a winning guess, we provide the statement to let the user know they've won. The full program up to this point is shown in Listing 2-4. Save your *HiLo.java* file and run the program to check that it works. It should prompt you to enter guesses until you guess the correct number.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        // Create a random number for the user to guess
        int theNumber = (int)(Math.random() * 100 + 1);
        // System.out.println( theNumber );
        int guess = 0;
        while (guess != theNumber)
        {
            System.out.println("Guess a number between 1 and 100:");
            guess = scan.nextInt();
            if (guess < theNumber)
                System.out.println(guess + " is too low. Try again.");
            else if (guess > theNumber)
                System.out.println(guess + " is too high. Try again.");
            else
                System.out.println(guess + " is correct. You win!");
        }    // End of while loop for guessing
    }
}
```

*Listing 2-4: The Hi-Lo guessing game is complete for a single full round of play.*

The full program is now a completely playable guessing game! After the user wins, the program tells them that they guessed correctly and won, and then it ends, as shown in Figure 2-6.


Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
Problems
Javadoc
Declaration
Console

```
<terminated> HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 6, 2016, 3:2
Guess a number between 1 and 100:
50
50 is too low. Try again.
Guess a number between 1 and 100:
75
75 is too low. Try again.
Guess a number between 1 and 100:
87
87 is too low. Try again.
Guess a number between 1 and 100:
93
93 is too low. Try again.
Guess a number between 1 and 100:
96
96 is correct. You win!
```

*Figure 2-6: One full play-through of the Hi-Lo guessing game— the program ends when the user guesses the secret number.*

Give yourself a hand! You've built a program in Java from scratch, and if this is your first program ever in Java, you deserve some kudos. Enjoy the game for a few rounds and see if you can guess the number in fewer tries each time. Test your program to make sure it works the way you want, and we'll add some improvements in the next section.

## *Adding a Play Again Loop*

Right now, the only way to play the guessing game again is to rerun the program in Eclipse. Fortunately, we already know there's a way to make our program do something over and over again—we need another loop!

The guessing game program ends when the user guesses the right number because there's nothing after the `while` loop. The `while` loop ends when the condition `(guess != theNumber)` is no longer true. A user might want to play over and over once they get the hang of the game. For this play again loop, we'll learn a new keyword and a new kind of loop: the `do-while` loop.

Like the `while` loop, a `do-while` loop repeats a block of statements as long as a condition is true. Unlike the `while` loop, however, the block of code inside a `do-while` loop is guaranteed to run at least once. There are times when the condition at the top of a `while` loop may be false before the loop even starts, so the entire loop and all the lines of code inside it are ignored. Think of the condition of a `while` loop as being like a thermostat on a heater. If the temperature of the room is already warm enough and the condition for the heater to turn on isn't met, the heater may not turn on at all.

For our guessing game, or almost any game program in general, we choose a `do-while` loop (we sometimes call this the *game loop*), because the user probably wants to play the game at least once. We also usually want to ask the user if they would like to play again, and the user typically responds yes or no (or `y` or `n` in a text-based game like this one). The game will continue to play through the game loop as long as the user responds with a yes.

To check the user's response, we'll need a new type of variable: a `String`. Strings are objects that hold text within double quotation marks, like `"y"`, or `"yes"`, or `"My name is Bryson! I hope you like my game!"`. Earlier we used an integer variable, or `int` type, to hold the numbers the user was guessing, but now we need to hold text, so we'll use a `String` instead. We can add a `String` variable to the top of the program, right after the `Scanner` setup:

```
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
```

Notice the `String` type begins with an uppercase `S`. This is because the `String` type is actually a class, complete with several useful functions for working with strings of text. I've named the variable `playAgain`, using camel case with a capital `A` to start the second word. Remember, no spaces are allowed in variable names. And, just like how we gave an initial value of `0` to the `guess` variable with `int guess = 0`, here we've given an initial value to the `playAgain` variable with `playAgain = ""`. The two double quotes, with no space between, indicate an empty string, or a `String` variable with no text in it. We'll assign a different text value to the variable later, when the user enters `y` or `n`.

Just as we did with the `while` loop, we'll need to figure out which statements should be repeated in the `do-while` loop. The `do-while` loop will be our main loop, so almost all of the statements in the program will go inside it. In fact, all the remaining statements after the `Scanner` and `String playAgain` statements will be contained in the `do-while` loop. Those steps describe one full round of play, so for each round, the game repeats all of those steps again, from choosing a new random number to declaring a winning guess and asking the user to play again.

So, we can add the do keyword and an opening brace immediately after these two lines and before the code that creates the secret number:

```
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber)
            {
```

Then, after the closing brace for the while loop for guessing and the brace following our last else statement, we'll ask the user if they would like to play again and get their response from the keyboard.

Then we need to close the do-while loop with a while condition to check whether the user replied with a yes:

```
        }   // End of while loop for guessing
          ❶ System.out.println("Would you like to play again (y/n)?");
          ❷ playAgain = scan.next();
      ❸ } while (playAgain.equalsIgnoreCase("y"));
  ❹ }
❺ }
```

The prompt asks the user "Would you like to play again (y/n)?" ❶, to which they can reply with a single letter, y for yes or n for no. At ❷, the scan.next() function scans the keyboard for input, but instead of looking for the next integer as nextInt() does, it looks for the next character or group of characters that the user types on the keyboard. Whatever the user types will get stored in the variable playAgain.

The line at ❸ closes the block of code that repeats the game with a brace, and it contains the while condition that determines whether the code will run again. Within the while condition, you can see an example of the equals() method of a String object. The equals() method tells you whether a string variable is exactly the same as another string of characters, and the equalsIgnoreCase() method tells you whether the strings are equal even if their capitalization is different. In our game, if the user wants to play again, they are asked to type y. However, if we just test for a lowercase y, we might miss an uppercase Y response. In this case, we want to be flexible by checking for the letter y, whether it is uppercase or lowercase, so we use the string method equalsIgnoreCase().

The final statement tells Java to keep do-ing the game loop while the string variable playAgain is either an uppercase or lowercase y. The final two closing braces at ❹ and ❺ are the ones that were already in the program.

The one at ❹ closes the `main()` method, and the one at ❺ closes the entire `HiLo` class. I've included them just to show where lines ❶ through ❸ should be inserted.

The complete game to this point is shown in Listing 2-5.

```
import java.util.Scanner;
public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber)
            {
                System.out.println("Guess a number between 1 and 100:");
                guess = scan.nextInt();
                if (guess < theNumber)
                    System.out.println(guess + " is too low. Try again.");
                else if (guess > theNumber)
                    System.out.println(guess + " is too high. Try again.");
                else
                    System.out.println(guess + " is correct. You win!");
            }   // End of while loop for guessing
            System.out.println("Would you like to play again (y/n)?");
            playAgain = scan.next();
        } while (playAgain.equalsIgnoreCase("y"));
    }
}
```

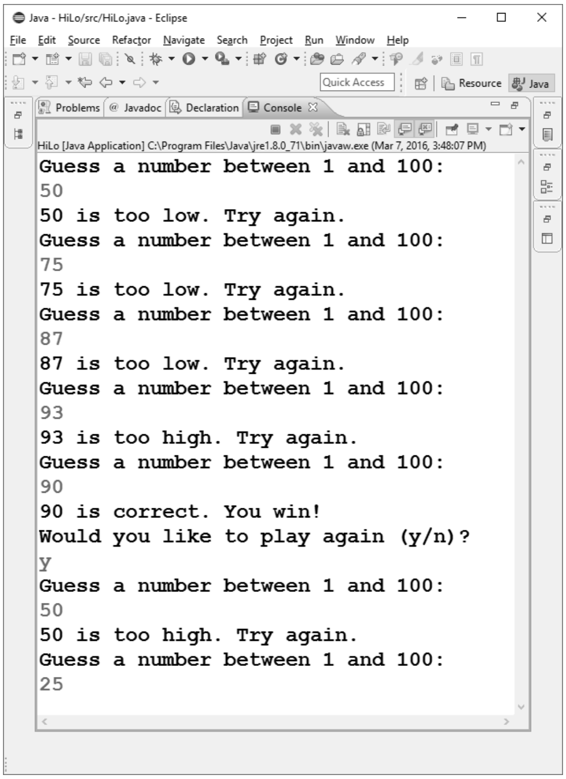
*Listing 2-5: The Hi-Lo guessing game is ready to play over and over again.*

Review your code to make sure you've added everything in the correct place, check your braces and semicolons, and take a moment to save your file. We'll test the game in the next section.

**NOTE** *Your indentation, which is the tab spacing at the beginning of each line, may not look exactly like the last code snippet because we've added braces in a couple of new places. Fortunately, adding new features, including loops and other blocks of code, is so common in Java that Eclipse has a menu option to clean up indentation automatically. First, select (highlight) all the text in your* HiLo.java *file on the screen. Then, go to* ***Source ▸ Correct Indentation****. Eclipse will correctly indent each line of code to show which statements are meant to be grouped together. As I mentioned before, the indentation doesn't matter to the computer (the program will run just fine even with* no *tabs or extra spaces), but good indentation and spacing help make the program easier to read.*

## Testing the Game

Now that the play again loop is in place, the game should run perfectly. First, save your *HiLo.java* file and choose **Run ▸ Run** to test the program. After you guess the first random number correctly, the program should ask you if you'd like to play again. As long as you respond **y** (or **Y**) and press ENTER, the program should keep giving you new random numbers to guess. In the screenshot in Figure 2-7, notice that the game starts over when I respond **y** to the prompt to play again.


Java - HiLo/src/HiLo.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Resource
Java
Problems
Javadoc
Declaration
Console

```
HiLo [Java Application] C:\Program Files\Java\jre1.8.0_71\bin\javaw.exe (Mar 7, 2016, 3:48:07 PM)
Guess a number between 1 and 100:
50
50 is too low. Try again.
Guess a number between 1 and 100:
75
75 is too low. Try again.
Guess a number between 1 and 100:
87
87 is too low. Try again.
Guess a number between 1 and 100:
93
93 is too high. Try again.
Guess a number between 1 and 100:
90
90 is correct. You win!
Would you like to play again (y/n)?
y
Guess a number between 1 and 100:
50
50 is too high. Try again.
Guess a number between 1 and 100:
25
```

*Figure 2-7: The guessing game is fully playable for multiple rounds as long as the user answers y or Y.*

When the user finishes playing and responds to the play again question with `n`, or anything other than `y` or `Y`, the game will end. However, we might want to thank them for playing after they've finished the game. Add the following line after the final `while` statement, before the final two closing braces:

```
        } while (playAgain.equalsIgnoreCase("y"));
        System.out.println("Thank you for playing! Goodbye.");
    }
}
```

Finally, the last line we'll add to the guessing game app is to address a warning you may have noticed in Eclipse. This warning appears as a faint yellow line under the declaration of the scan object, as well as a yellow triangle with an exclamation point to the left of that line. Eclipse is bringing to our attention that we've opened a resource that we haven't closed. In programming, this can create what's known as a *resource leak*. This doesn't usually matter if we just open one Scanner object for keyboard input, but if we leave multiple Scanner objects open without closing them, the program could fill up memory, slowing or even crashing the user's system. We use the close() method of the Scanner class to tell our program to close the connection to the keyboard.

Add the following line after the println() statement thanking the user for playing, before the final two closing braces:

```
        System.out.println("Thank you for playing! Goodbye.");
        scan.close();
    }
}
```

You'll notice that the yellow warning disappears from the Eclipse editor window when we add this line. Eclipse helps with common programming errors like misspellings or missing punctuation, and it even warns us about problems that *could* occur like resource leaks and unused variables. As you build bigger, more complex applications in Java, these features of the IDE will become even more valuable. You can find more information on using Eclipse to debug your programs in Appendix B.

The finished program, shown in Listing 2-6, is a fully playable guessing game, complete with the option to play again and guess a new random number every game.

```
import java.util.Scanner;

public class HiLo {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String playAgain = "";
        do {
            // Create a random number for the user to guess
            int theNumber = (int)(Math.random() * 100 + 1);
            // System.out.println( theNumber );
            int guess = 0;
            while (guess != theNumber)
            {
                System.out.println("Guess a number between 1 and 100:");
                guess = scan.nextInt();
                if (guess < theNumber)
                    System.out.println(guess + " is too low. Try again.");
                else if (guess > theNumber)
                    System.out.println(guess + " is too high. Try again.");
                else
                    System.out.println(guess + " is correct. You win!");
            }   // End of while loop for guessing
            System.out.println("Would you like to play again (y/n)?");
            playAgain = scan.next();
        } while(playAgain.equalsIgnoreCase("y"));
        System.out.println("Thank you for playing! Goodbye.");
        scan.close();
    }
}
```

*Listing 2-6: The finished text-based, command line guessing game*

There are a few things worth noting about the finished Hi-Lo guessing game program. First, despite all the work in writing it, the code is relatively short—fewer than 30 lines long. Nevertheless, you could play it forever if you wanted to. Second, this program not only demonstrates conditions and looping, it also makes use of a loop inside another loop. This is called a *nested loop*, because the guessing loop is contained in, or nested inside, the play again loop. The indentation helps us see where the `do-while` loop begins and ends, and we can see the smaller `while` loop and its `if` statements tabbed over and nested inside the bigger `do-while` loop. Finally, we end the program neatly—both for the user, by thanking them for playing, and for the computer, by closing the scanner resource.

## What You Learned

While building a simple, fun, playable game, we've picked up several valuable programming concepts along the way. That's the way I first learned how to code as a kid—I would find a fun game or graphical app, program it, then change it, take it apart, and try new things. Play and exploration are an important part of learning anything new, and I hope you'll take a little time to try new things with each program. The Programming Challenges at the end of each chapter will also give you an opportunity to try a few new things.

In building this guessing game, we've developed a wide range of skills in Java:

- Creating a new class, `HiLo`
- Importing an existing Java package, `java.util.Scanner`
- Using a `Scanner` object to accept keyboard input
- Declaring and initializing integer and string variables
- Generating a random number with `Math.random()` and casting it to an integer
- Using `while` and `do-while` loops to repeat a set of steps while a condition is true
- Printing text strings and variable values to the command line console
- Scanning integers and strings from the keyboard and storing them in variables

- Testing various conditional expressions in `if` and `if-else` statements
- Using `String` methods to compare string values with `equalsIgnoreCase()`
- Closing input resources like `Scanner` objects with the `close()` method
- Running a command line program from inside Eclipse

In addition to practical skills, you've also developed a working knowledge of several important programming concepts in Java:

**Variables** `theNumber` is an integer variable, or `int`, and so is `guess`. `playAgain` is a string variable, or `String`. We change the values of these variables as we play the game by entering new number guesses or answering `y` or `n`.

**Methods** Methods are what we call functions in Java. `Math.random()` is a method for generating random numbers between 0.0 and 1.0. The `scan.nextInt()` method accepts numeric input from the user. `System.out.println()` is a function for displaying text to the console or terminal window.

**Conditionals** The `if-else` statements allow us to test whether a condition, like `guess < theNumber`, is true and run a different block of code depending on the outcome of that test. We also use conditional expressions to determine whether to perform a loop again, as in the statement `while (guess != theNumber)`. This statement will loop as long as `guess` is not equal to `theNumber`. Remember the test for "is equal to" is the double equal sign: `==`.

**Loops** A `while` loop lets us repeat a block of code as long as a condition is true. We used a `while` loop in the guessing game to keep asking the user for another guess until they got the right number. A `do-while` loop always runs at least once, and we used one to ask the user if they wanted to play again.

**Classes** The whole HiLo app is a Java class, `public class HiLo`. A class is a template. Now that we've built a class template for the HiLo guessing game, we can reuse it to play the guessing game across many different computers. We also imported the `Scanner` and `Math` classes in this app to accept user input and generate random numbers. We'll write our own classes to do something new, and we'll take advantage of the classes already included in Java to do everyday tasks like input, math, and more.

## Programming Challenges

Try these programming challenges to review and practice what you've learned and to expand your programming skills. If you get stuck, you can visit the book's website at *http://www.nostarch.com/learnjava/* to download sample solutions, or you can watch this lesson in the video course online at *http://www.udemy.com/java-the-easy-way/* for step-by-step solutions. Chapter 2 is free to preview, and you can use the coupon code BOOKHALFOFF to save 50 percent when you buy the full course.

## #1: Expanding Your Range

For this first programming challenge, change the guessing game to use a bigger range of numbers. Instead of 1 to 100, try having the user guess between –100 and 100! (Hint: Multiply `Math.random()` by 200 and subtract 100 from the result.)

Remember to change *both* the programming statement that generates the random number *and* the prompt that tells the user the range they should guess between.

If you want an easier game, you can change the range from 1 to 10 and wow your friends when you can guess the secret number in just four tries. Try other ranges, like 1 to 1,000, or even 1 to 1,000,000, or use negative ranges if you want! (Remember that you can't use commas when writing the number in Java.) You'll not only get better at programming, but you might also improve your math skills. Change the program however you'd like and have fun with it!

### STRATEGIZE YOUR GUESSES

You may discover that the more you play the guessing game, the faster you'll be able to guess the secret number. You might even stumble onto the fact that you can guess the number fastest by guessing in the middle of a range with each new guess. This technique is called a *binary search*. Guessing a number in the middle of the possible range cuts the number of possibilities in half each time.

Here's how it works. For a number from 1 to 100, guess 50. If that's too low, you know the secret number must be between 51 and 100, so guess 75, right in the middle of this range. If that's too low, try a number halfway between 76 and 100, which would be 87. One reason the binary search is so valuable is that we can reduce the number of guesses to just seven tries (or fewer) to find a secret number between 1 and 100, every time. Try it out!

When you get the hang of guessing a number between 1 and 100 in seven tries or less, try guessing a number from 1 to 1,000 in just 10 tries. If you're really brave (and have a pencil nearby), try guessing a number from 1 to 1,000,000. Believe it or not, it should take you just 20 guesses.

## #2: Counting Tries

We've already built a pretty cool guessing game app, but let's try adding one more feature to the game. Your challenge is to count and report how many tries it takes the user to guess the secret number. It could look something like the following:

```
62 is correct! You win!
It only took you 7 tries! Good work!
```

To accomplish this task, you'll need to create a new variable (you might add a line like `int numberOfTries = 0;`), and you'll have to add to the number of tries every time the guessing loop executes. You can do this by increasing the variable `numberOfTries` by one for each new loop using `numberOfTries = numberOfTries + 1`. Be sure to include text to let the user know the number of tries.

It may take a few tries to get all the code working in the right order at the right time, but it's worth the effort and will help you practice your new skills. In the next chapter, we'll build this feature into a different version of the guessing game. In the meantime, I hope you'll come up with even more ideas for improving and changing the game. Playing with your programs, taking them apart, and rebuilding them can be the best way to learn.

## #3: Playing MadLibs

For your final challenge in this chapter, let's write a completely new program. We've learned how to ask a user for input and store it in a variable. We've also learned how to print out both text and variable values to the screen. With those skills, you can build even more interesting and fun programs.

Have you ever played MadLibs? Let's try to use our new skills to build a program in that same style. MadLibs ask a player for various words or parts of speech, such as a color, a past-tense verb, or an adjective, and then insert the words the player chose into a template, usually resulting in a funny story. For example, if a player gave a color of "pink," a past-tense verb of "burped," and an adjective of "silly" and then inserted them into the template "The ____ dragon ____ at the ____ knight," we would get the result "The *pink* dragon *burped* at the *silly* knight."

Now, the challenge is to write a new program, *MadLibs.java*, with a class called `MadLibs` and a `main()` method that prompts the user for several words. Those words should each be stored in a different `String` variable, like `color`, `pastTenseVerb`, `adjective`, and `noun`, which you initialize as empty strings. Then, after the user has entered their last word, the program should print a completed sentence or story by replacing the empty strings with the words the user provided, like this:

```
System.out.print("The " + color + " dragon " + pastTenseVerb + " at the " + adjective);
System.out.println(" knight, who rode in on a sturdy, giant " + noun + ".");
```

Note that the first statement is a `print()` statement instead of a `println()`. The `print()` statement continues printing at the end of the same line, allowing us to build a longer paragraph or story. The `println()` statement, however, always skips a line after printing, like when you press ENTER at the end of the line. You can write a longer MadLibs story by using different variable names like `noun1`, `noun2`, and `noun3`. Give it a try, and get ready to laugh at the funny stories you create! Try to personalize each program you create by adding new features and making it your own.