Lesson 2: Writing Your First Java Program

Introduction

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."

—Linus Torvalds, software engineer, hacker, and principal creator of the Linux operating system

As you can tell from the quotation above, I think programming is fun. It's often challenging, often frustrating, but I've enjoyed it the whole time I've been doing it. I hope that you find it as much fun as I have over the years.

For our first Java program, we're going to follow tradition. Even in a field as new as computer programming, there are traditions. One of them is that someone's first program should display a "Hello, world!" message, so that's what ours will do.

We're going to take two runs at this program. The first is the simplest way I know to write the program, and we'll go through it quickly. The second approach (in Lesson 3) will use Java's object-oriented program structure to produce the same output, and we're going to go through that version in more detail.

Along the way, you'll learn a fair amount about Java's *syntax* (the rules about the format a program has to take), its data types, and its naming conventions.

So if you're ready, let's jump in . . .

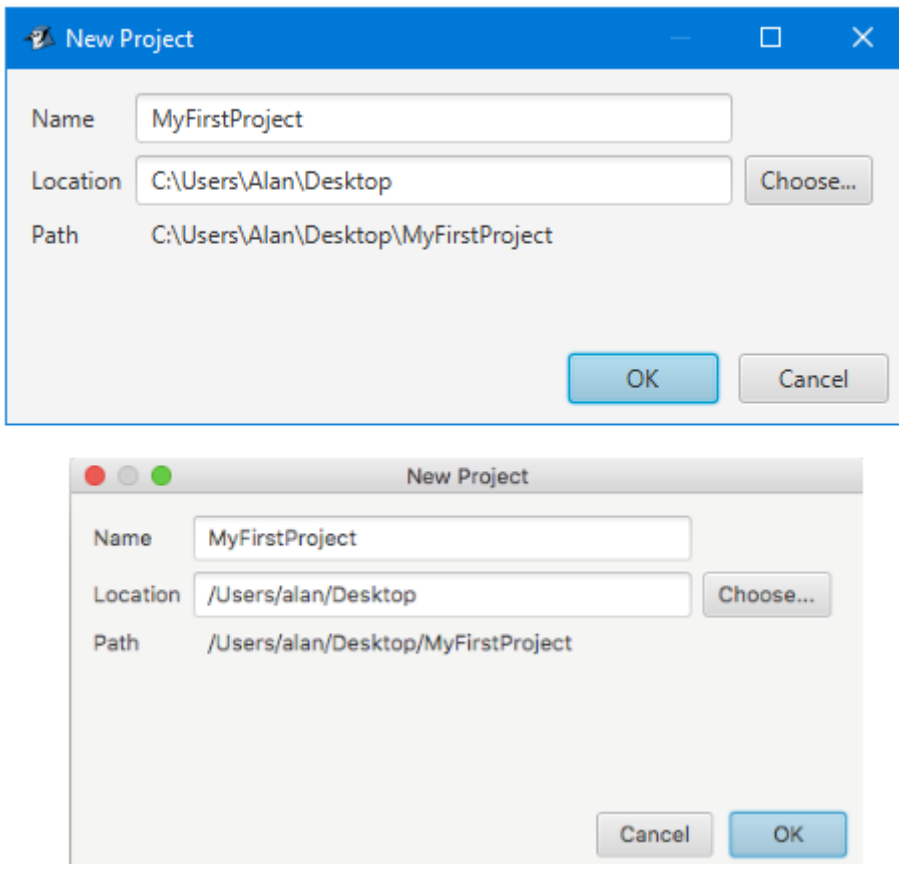
Writing and Compiling a Program

First, start BlueJ by double-clicking its icon. If it opens with a project still active, you can close it using the Project menu's **Close** option.

Now go to the Project menu and select

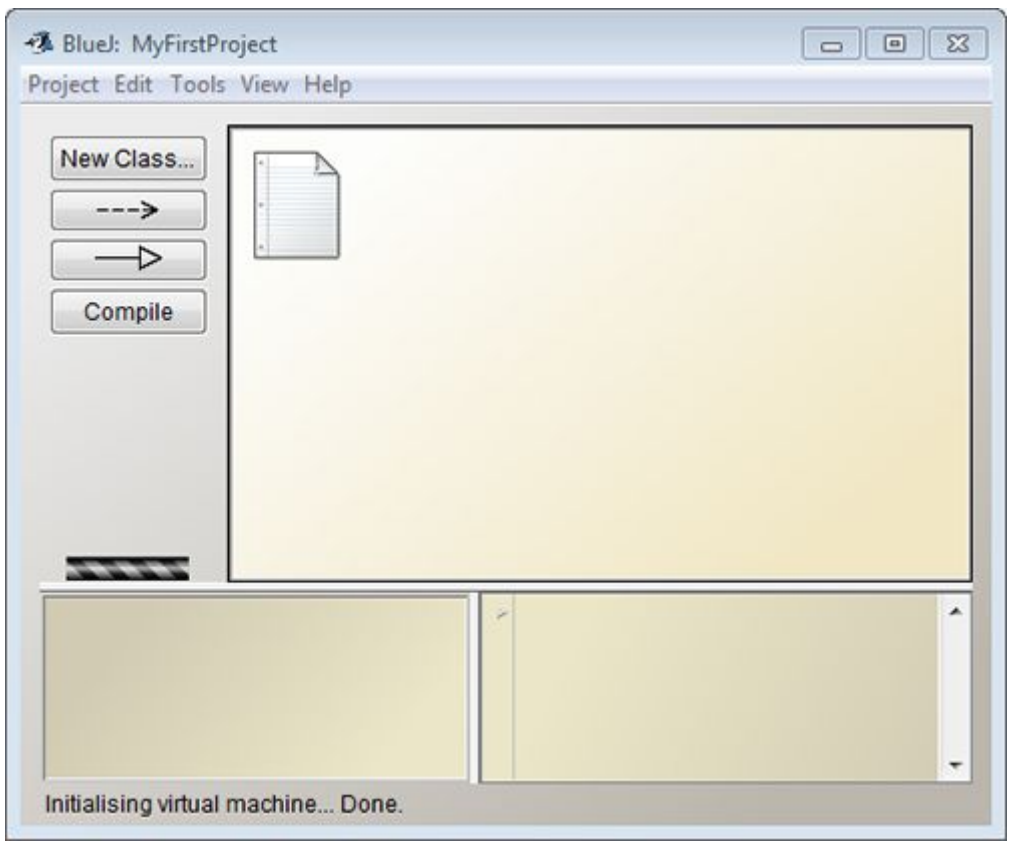
New Project

. BlueJ will open a New Project dialog box asking you which directory to put your project in. To keep things simple you can name the project MyFirstProject and put it on your desktop. The image below shows how that looks in Windows and on a Mac (though your username probably won't be Alan).



Creating a new BlueJ project in Windows and in Mac OS X

Click **OK** after specifying your project name and the folder in which to store it. Bluejay will open with an empty project open. You'll see an icon for the README file that BlueJ automatically puts in every project. It looks like a sheet of paper, and you can use it to keep notes for yourself or other programmers as you develop your project. But we won't be using that just yet. In your BlueJ project window, your new project should look something like this:



A new BlueJ project

The icon in the top-left corner of the project window is a README.TXT file that BlueJ automatically creates for you. You can use it later for documentation or other text information you want to include.

To create your program, click BlueJ's **Edit** menu, and choose **New Class**, or click the **New Class** button at the top left. In Java, a *class* is like a blueprint that you use to create different objects. A dialog box will open, asking you to name your class and select its type. Enter the name *HelloWorld*, leave the class type as it is (the default is Class), and click **OK.**

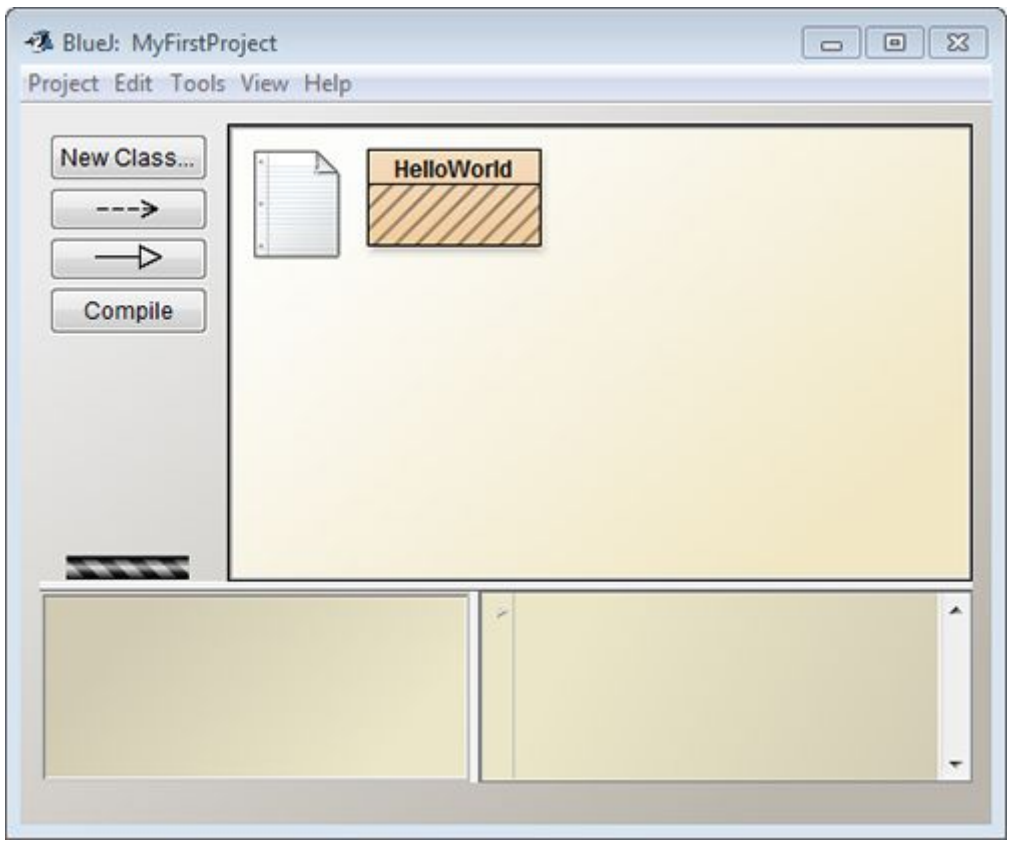
Java's Rules for Naming Things

When you name a Java class, use a noun, since Java's classes represent entities, or things. I'm taking a little liberty with the name this time, since the traditional name for a first program is HelloWorld.

Another convention is to capitalize the first letter of each word in the name. This method of omitting spaces between words and using a capital letter to start each word is called *Pascal case*. It's similar to another convention called *camel case* but slightly different. With camel case, the very first letter is always lowercase. As you'll see, Java uses camel case for some names. But class names are Pascal case.

You can't include spaces in Java names, but letters, numbers, and underscores (_) are valid. Some examples of Java class names from the Java libraries are *ArrayList*, *DirectoryManager*, *StandardCopyOption*, and *ArrayListOutOfBoundsException*. While Java will accept underscores in class names, they're rarely used there. We'll see examples of names with underscores later.

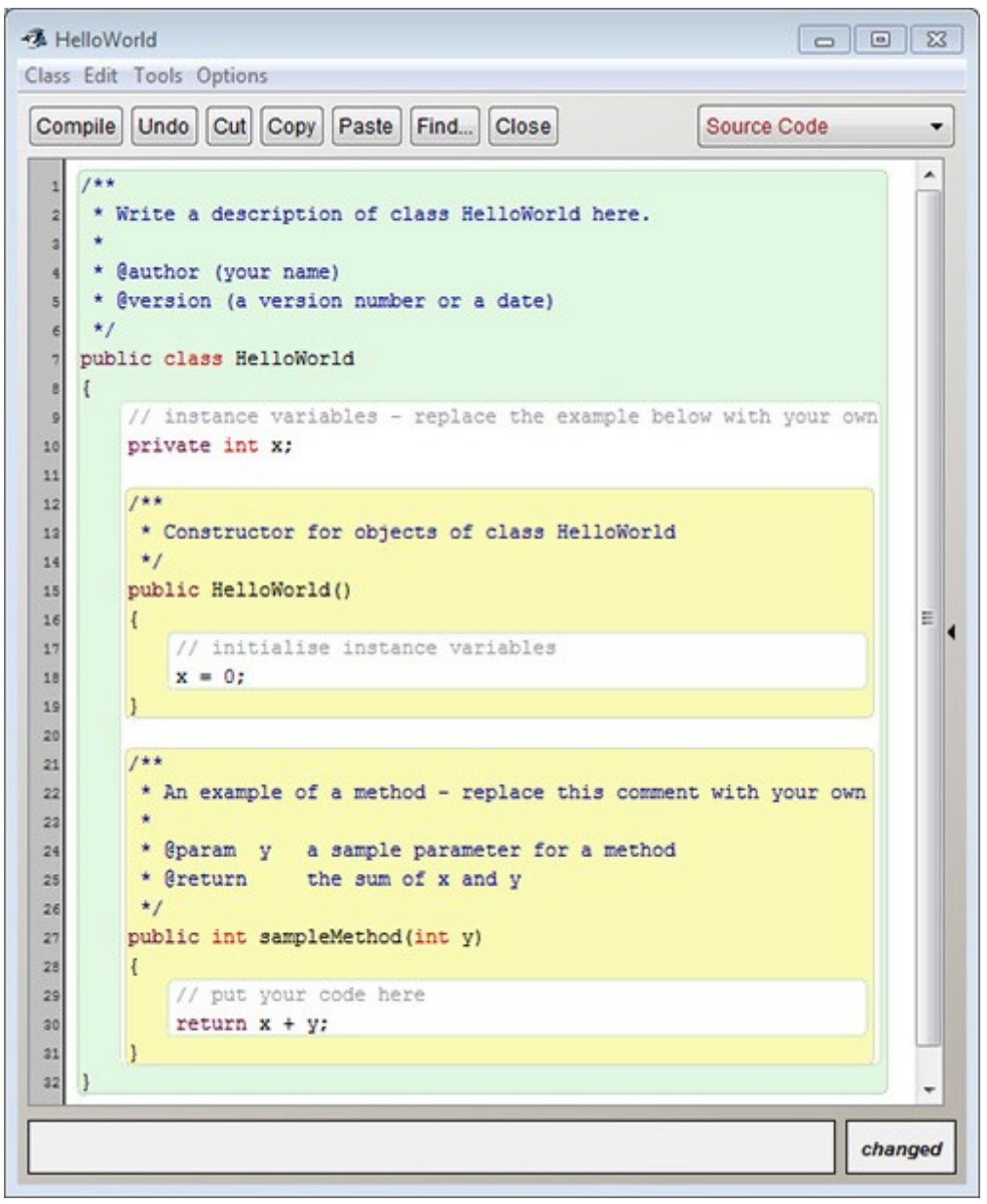
Once you click OK, a rectangular icon with the name you chose appears in your BlueJ window.



HelloWorld class icon

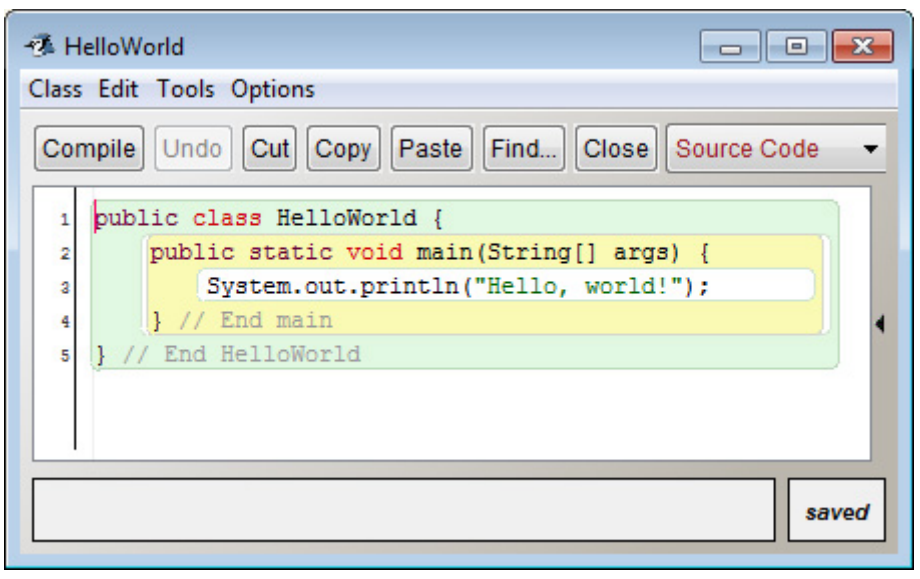
The icon represents the class that you're creating. The diagonal lines tell you that the class icon hasn't been compiled yet. BlueJ has created a source code file for you—it's in the project folder, and its name is *HelloWorld.java*. When you create a new class in Java, its source code file must have the same name as the class, or the Java compiler won't accept it.

Now you're ready to edit the source code file and make it the program we want it to be. To open the source code in the editor, double-click the **HelloWorld** box. The editor will open and will look like this:



BlueJ's text editor window

This window is BlueJ's text editor, open to your new program file. BlueJ generated some default code, but we'll delete that and write our own. So select all the code in the editor window by using your mouse to highlight it all, and delete it. Then let's write a very short Java program.



HelloWorld code

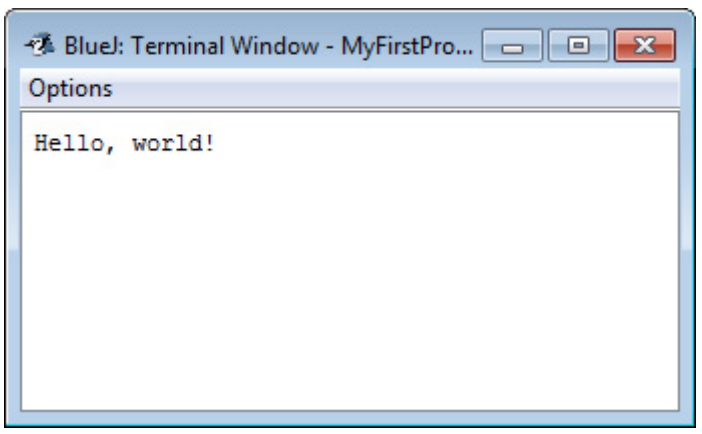
Copy and paste or type this text into the empty editor window.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    } // End main
} // End HelloWorld
```

Click the **Compile** button at the top-left corner of the window. If you've typed everything correctly, you'll see a message that says something like, "Class compiled—no syntax errors."

If you see an error message in that box, you may have typed something incorrectly, or perhaps you didn't copy and paste the entire code. Compare the code to what you see above, and see if you can find the problem. Then try compiling again. If you can't find the problem but it still won't compile, post the code in a message in the Discussion Area.

If your program compiled without errors, then go back to the main BlueJ window, right-click the HelloWorld icon, and run the program by selecting the **void main(String[] args)** option from the drop-down menu. (If you're a Mac user, you'll press CONTROL as you click instead of right-clicking.) The result should look like this:



Congratulations! You've just written and executed a Java program.

You might wonder why some of the words are different colors. Let me explain BlueJ's color-coding.

- **Keywords:** These are words that have special meaning in Java and that you can't use for any other purpose, such as names. BlueJ color-codes most of Java's keywords in a dark red font—for example, public and static.

- **Data types**: These are the different information formats that Java uses. Java data types are brighter red, as you can see in class and void.

- **Text strings:** A text string is any text in a program that you want to appear as text in Java. To let Java know that the words are text rather than keywords, data types, or other information that Java uses, you enclose the text in double quotation marks. BlueJ makes text strings green, like the string "Hello, world!".

BlueJ puts everything else in black text.

Learning a Programming Language

Tip

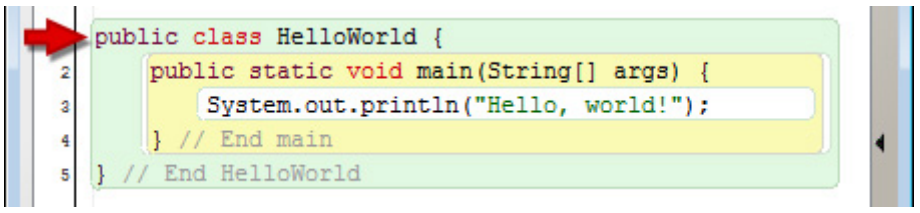
Unless you already know another programming language—and maybe even if you do—a lot of the explanation in this lesson may not make sense at first. If that's the case, don't feel bad! I was in a fog for quite a while learning my first programming language, and that happened again when I started digging into Java.

Most new programmers (and even a lot of experienced ones) feel confused when first encountering a new language. It takes some time, but I'll explain things as clearly as I can, and the terms and names will start to make sense as you gain experience. Please feel free to share your questions and thoughts with other students in the Discussion Area.

Now let's take a closer look at our new program.

Reviewing Our New Program Line by Line

Here's the first line of the program.




```
public class HelloWorld {
```

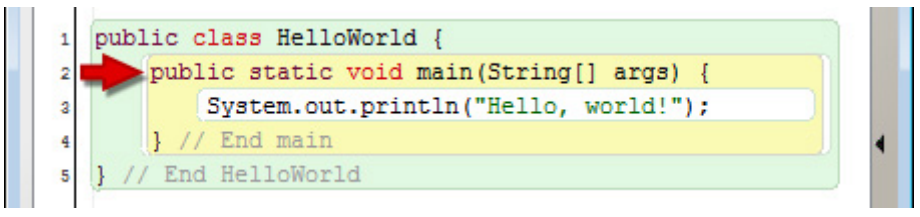
This line *declares* a new class to Java; that is, it tells Java that we want to create a new class named HelloWorld. We're not going to go over every detail at this point, but the line tells Java all these things:

- It's a class (as opposed to other things we can declare).
- The class is public (anyone can use it).
- The name of the class is HelloWorld.

Everything after the left curly bracket ({) and before the matching right bracket (}) that appears later in the code is going to be part of our new class.

Every program in Java has to be contained in a class. Not every class is a complete program, though. Many (maybe most) Java classes aren't programs in their own right; instead, people designed them for use with other classes. In Lesson 3, you'll see an example of a class that's not a program.

The second line of our program is:




```
    public static void main(String[] args) {
```

This line tells Java that we want to declare a method named "main" inside our HelloWorld class. Everything after this left curly bracket ({) and before its matching right bracket (}) will be part of our main method.

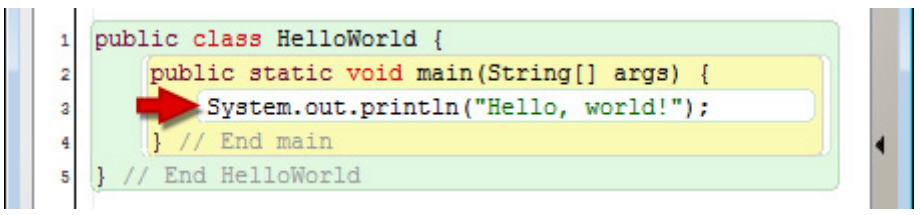
Methods are Java's step-by-step instructions or procedures. If you know another programming language, you can think of methods as similar to what some languages call functions or subroutines. In Java, any name followed by a pair of parentheses is a method name, whether or not the parentheses have anything inside them.

A class with a *main method* is a stand-alone program that the Java Virtual Machine (JVM) can execute. (As we discussed in Lesson 1, Chapter 5, the JVM is the part of a computer or other device that allows it to run Java programs.)

What if a class doesn't have a main method? Then it can't be executed on its own. Its purpose is to be used within other classes.

The main method in any class is where the JVM starts when it runs a program. It's the top-level logic for the program. And it has to be declared exactly as you see it in the above line of code.

The third line of our program looks like this.




```
        System.out.println("Hello, world!");
```

This line produces our program's output. It tells Java to do these things:

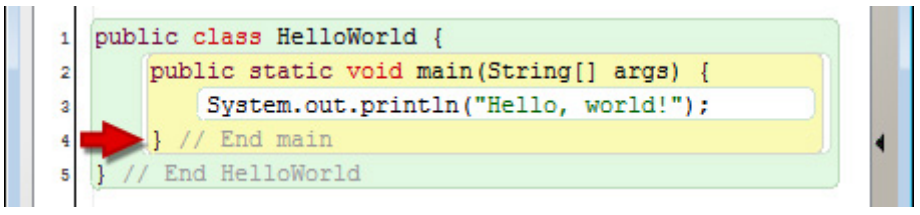
- Go to the System class (remember, class names start with uppercase letters).
- Find a variable in that class named out that connects to the default output device.
- Call a method named println that belongs to that variable.

See the periods between those words? That's *dot notation* in Java. When you put periods between names, that means that the name on the right side of the dot is part of the name on the left side of the dot. Java puts all those things together into a series of steps (as I described in the bulleted list above) rather than looking at each word as a separate command.

The println() method causes Java to take the information we gave it and put it onscreen. How does it know what to display? The last part of the line, within the parentheses, contains the information we want to put onscreen. Java programmers call that information the *argument* of the call.

The line ends with a semicolon. Most statements in Java end in either a semicolon or a left curly bracket. The class and method declarations ended with left curly brackets, so they don't need semicolons.

Let's move on to the fourth line.

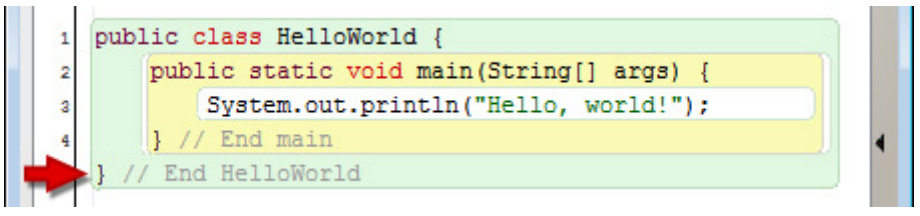



```
} // End main
```

This line has the right curly bracket that matches the left curly bracket at the start of our main method. That means this line is the end of the main method. (Right curly brackets don't need semicolons after them.)

The double slash marks indicate that there's a comment in this line. Java ignores comments when it's compiling the program. Comments are for us human readers, to help explain what's going on in the program. Everything from the slashes to the end of the line is a comment. In this case, I included a comment to remind me what the bracket is ending.

Now here's the fifth line.




```
} // End HelloWorld
```

Like line 4, this line has a right curly bracket and a comment. In this case, the bracket matches the opening bracket for the class, so this closes the class contents. Our program has two sets of right curly brackets, which might be confusing for us but not for Java. The comment indicates that we've reached the end of the HelloWorld class.

So that's our first program! Feel free to play with it to make it generate different lines or multiple lines.

Try this true-false activity to see how well you remember some of the ideas we covered in this chapter.

Text equivalent start.

Instructions: Read each statement and decide whether it's true or false. Then read the second column for the correct answer and explanation.

Statement	Answer with Explanation
Most statements in Java end with a period (.) or a right bracket (]).	False. Actually, most statements in Java end with a semicolon (;) or a right curly bracket (})
In Java, comments are for people to read and for computers to ignore.	True. That's right! In Java, you indicate a comment with a double slash mark (//).
Using dot notation tells Java to consider commands one by one rather than connecting them into a series.	False. Dot notation tells Java to consider commands as a series of steps.

Text equivalent stop.

How'd you do? Are these terms starting to become a little more familiar to you?

Java's Programming Conventions

Let's look at a couple of conventions that Java programmers have developed over the years. These formatting practices make the code easier for us humans to read.

Comments: Single-Line and Multi-Line

Like every other programming language I'm familiar with, Java allows programmers to add comments in the code. And like every other programming teacher, I encourage you to use lots of comments in your code so that when someone else reads it, it'll be easy for that person to sort out what's going on.

Java supports two styles of comments.

The first type is a *single-line comment*. This type may take only part of a line, but it never runs longer than one line. It begins with a double slash (//), and Java treats everything from the double slash to the end of the line as a comment.

We've already seen a couple of examples of this type, but here are some more:

```
// This comment takes a whole line
            // This comment starts in column 13
    int x = 0;  // This comment follows some code on the same line
```

All three of these are valid comments in Java.

Take a look at the third example. For this one, Java will execute the code at the start of the line, and then it'll ignore the comment beginning with the slashes. So you can't put any executable code after a comment of this type on the same line, since Java will ignore it.

You can also use a *multi-line comment* or *block comment*, which you can spread across any number of lines. This comment starts with the characters /* and continues until Java sees the characters */. Here are a couple of examples:

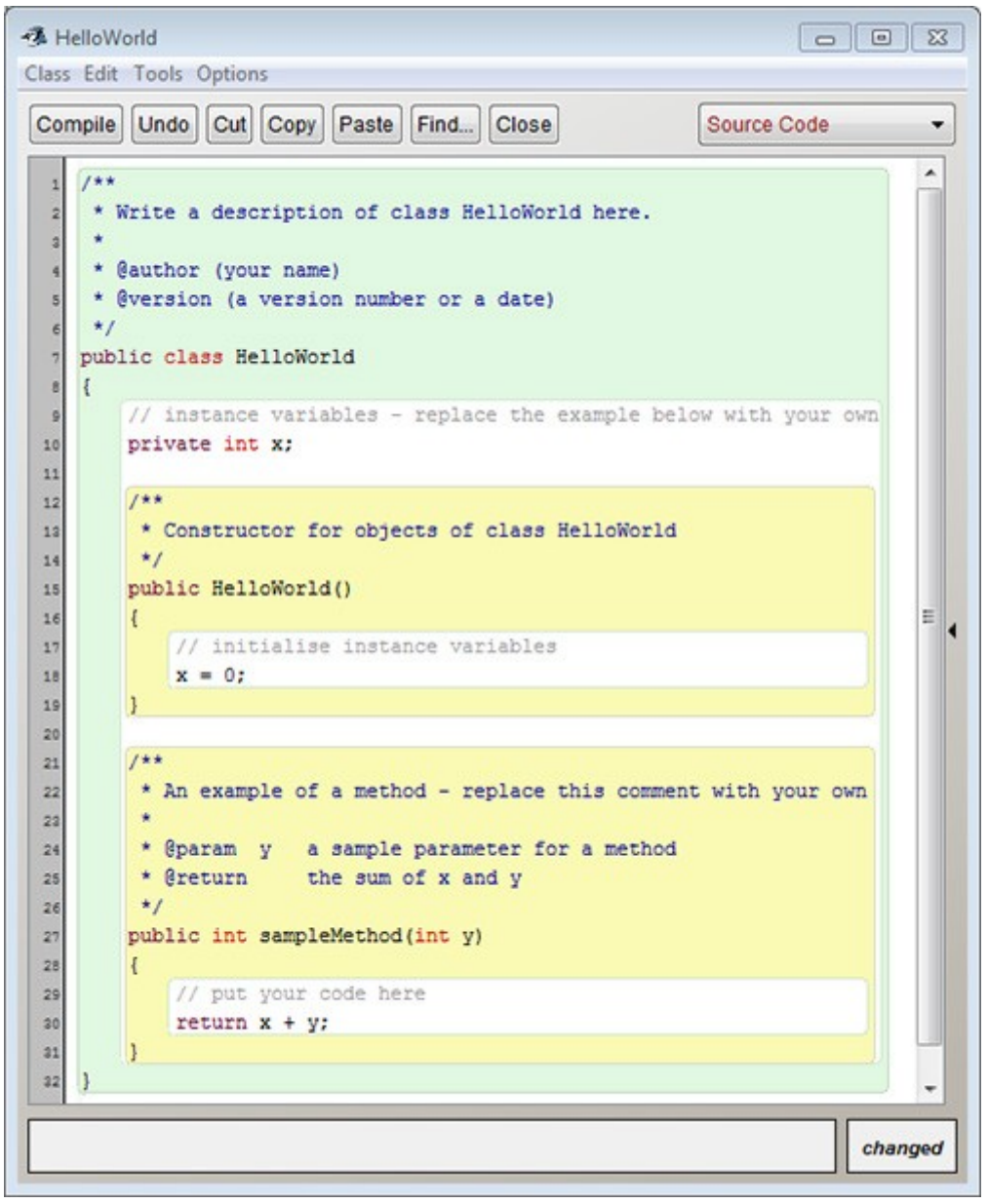
```
/* This comment takes part of a line */


/* This comment
 * is spread
 * across several lines
 */
```

Since these comments have a specific end point, you can put code after them in the same line, but most programmers don't because it makes the code harder for someone else to read and figure out.

BlueJ highlights single-line and multi-line comments in gray.

There's also a "documentation" type of comment that's a subset of the multi-line comments. This type of comment starts with /** instead of /*. Oracle has a program named Javadoc that reads the source code of a program and uses any documentation comments in the code to generate documentation.

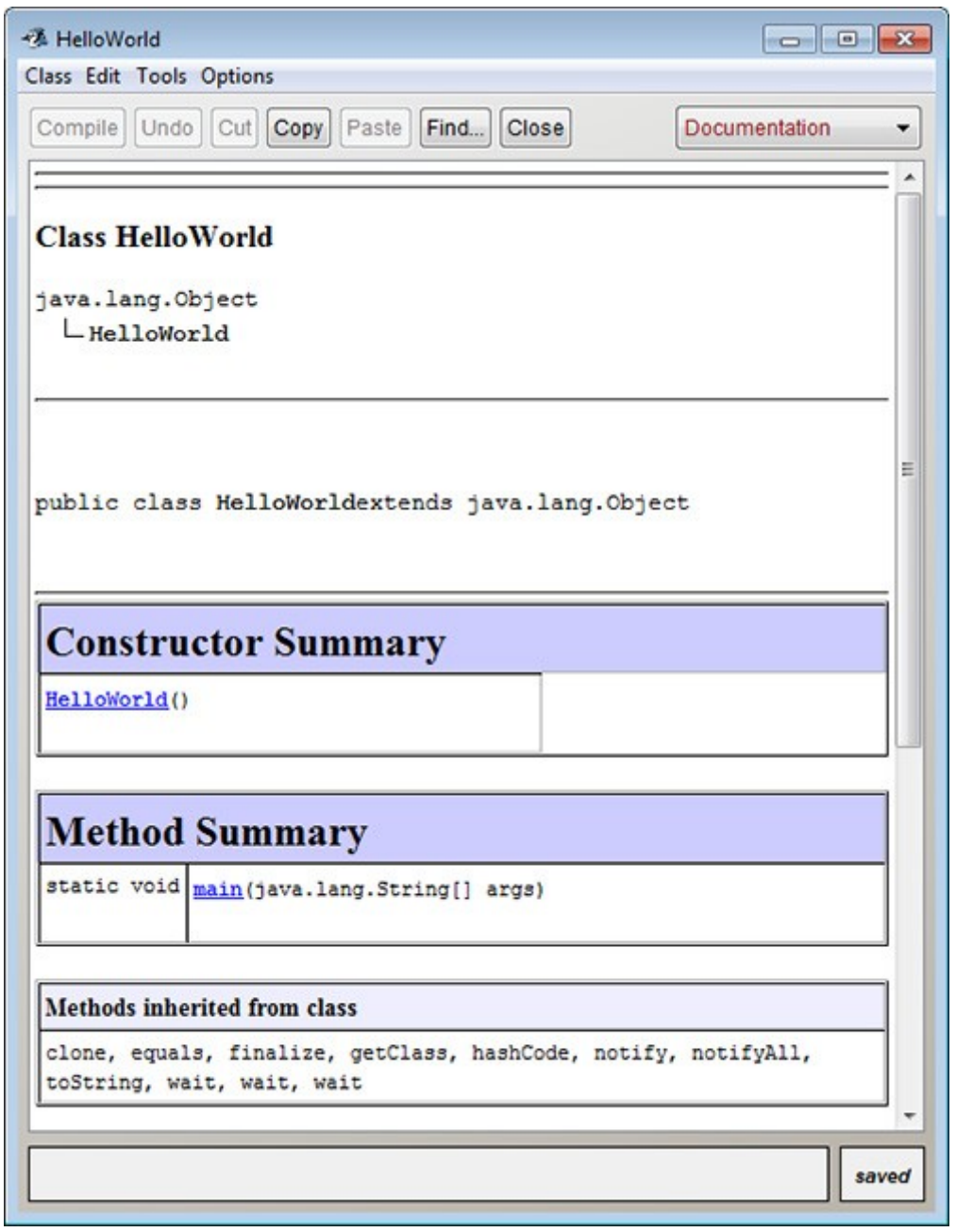
If you look at the editor window I showed you earlier, you'll see three documentation comments that BlueJ generated. I'll reproduce that editor window for you here so you don't have to go back.



BlueJ's text editor window

BlueJ lets you look at how Javadoc works. If you go to the top-right corner of BlueJ's editor window, click the drop-down menu there, and select **Documentation**, BlueJ will run Javadoc on the source code and show you what the documentation looks like.

Here's the documentation that Javadoc generated from the comments in that first editor window:



Javadoc output

If our session had documentation comments, you'd see them here. You'll see this type of documentation frequently.

BlueJ colors documentation comments blue so they stand out when you're reading the code.

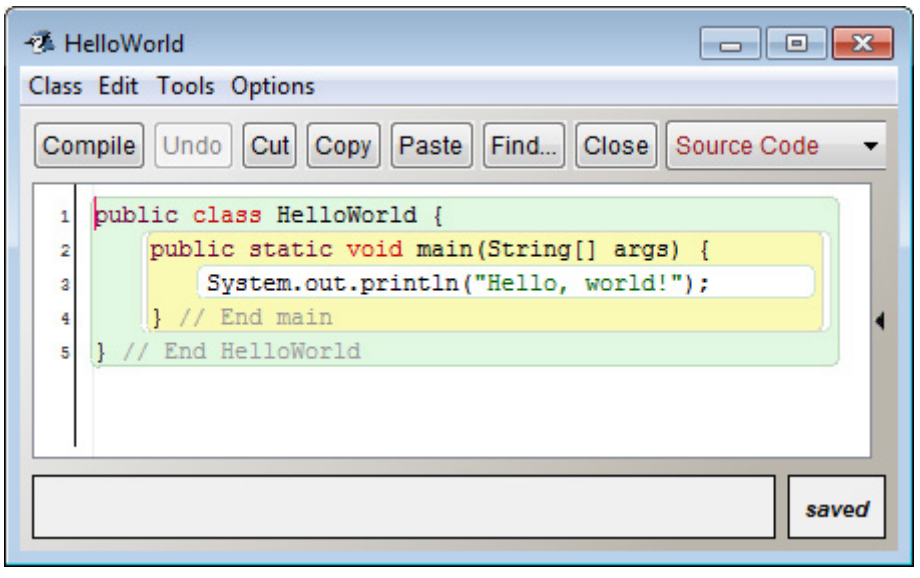
Java Code Formatting: Indents and Curly Brackets

Many code format conventions are from earlier languages like C and C++. For instance, indenting a program makes it much easier to read once you get used to it.

Take a look back at the images of the editor window. In both cases, the class declaration started in column 1. Method declarations were indented four spaces, and code within the methods was indented four more spaces. There's nothing magic about the number four; it's just widely used.

If you want BlueJ to indent your code for you, go to the **Edit** menu and choose **Auto-layout**. Or you can hit the CTRL + SHIFT + I key combination in Windows, or COMMAND + SHIFT + I on a Mac, and BlueJ will try to indent the code correctly. It's not perfect, but it's a quick way to format code.

Another coding convention has to do with placing opening curly brackets. If you look at that first editor window again, you'll see that each opening bracket is on its own line. In the second window, they're at the end of the preceding line. Here's that second window again.



The second editor window

Both ways of using curly brackets are common, so you can use either method.

Now let's discuss about how to name things in your Java programs.

Formatting Names in Java

Java's name-formatting convention is so widely used that if you don't follow it, people will probably complain to you about it. So here's how the conventions work.

A Java name can be any combination of letters, numbers, underscores, and dollar signs, as long as it doesn't start with a number. Spaces aren't allowed, and certain patterns have become so widely used that people expect to see them.

Here are some tips to help you use names correctly in your Java programs:

- **Don't start names with dollar signs or underscores.** Those are usually reserved for system names, not document names.

- **Use descriptive names.** A name should describe the document's purpose, so use names like accountNumber, firstName, and aLongJavaVariableName rather than x, y, and z. You'll have to type more, but it'll be beneficial in the long run.

- ***Variable names* are nouns that describe the data they will hold.** They should start with a lowercase letter, and each word after the first should start with an uppercase letter. So accountNumber, firstName, and aLongJavaVariableName are all good variable names.

- ***Method names* are verbs that describe the action they provide.** They should follow the same formatting rules as variable names, but method names always have parentheses after them. Some examples are setColor() and getText().

- ***Constant names* are for values that can't change.** These are nouns, and they're all uppercase. Separate words with underscores. Java constant names look like this: PI, MAX_VALUE, and MIN_VALUE.

- ***Class names* are nouns that describe the entity they represent**. They start with an uppercase letter, and all words after the first also start with uppercase letters. String, Math, and Graphics are class names that we'll use in this course.

Text equivalent start.

Instructions: Read each name in the first column, and decide which letter is incorrectly formatted. Then read the second column to see the correct answer and explanation.

Name	Answer with explanation
MAX_vALUE	v. Constant names should be all uppercase.
accountnumber	n. Each word after the first should start with an uppercase letter in a Variable name.
LastName	L. Variable names should start with a lowercase letter.
aLongname	n. Each word after the first should start with an uppercase letter in a Variable name.
MIN VALUE	space. Separate words with underscores in a Constant name.
SetColor()	S. Method names should start with a lowercase letter.
first Name	space. Spaces aren't allowed in any variable names.
getText(	space. Method names should always have an open and close parentheses after them.

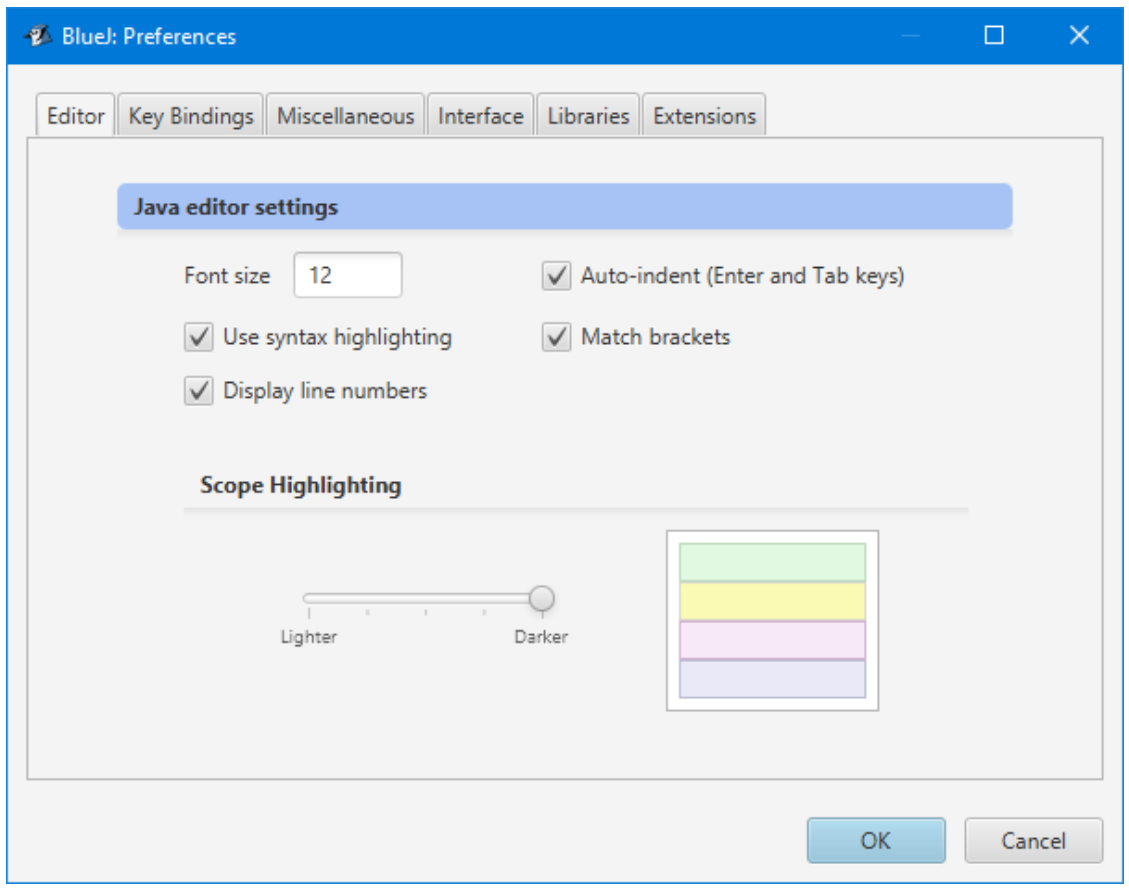
Text equivalent stop.

I know that seems like a lot of rules to follow! But as you use them, they'll become second nature.

We have a couple of topics still to cover, and then we'll wrap up this lesson.

Using BlueJ's Preferences and Terminal Window

When you're in BlueJ, on a Mac you can choose BlueJ > Preferences to change settings. In Windows, choose Tools > Preferences from the menu bar. The Editor tab shows options for styling the code editor. Here's how my BlueJ editor preferences are set.



BlueJ Editor Preferences

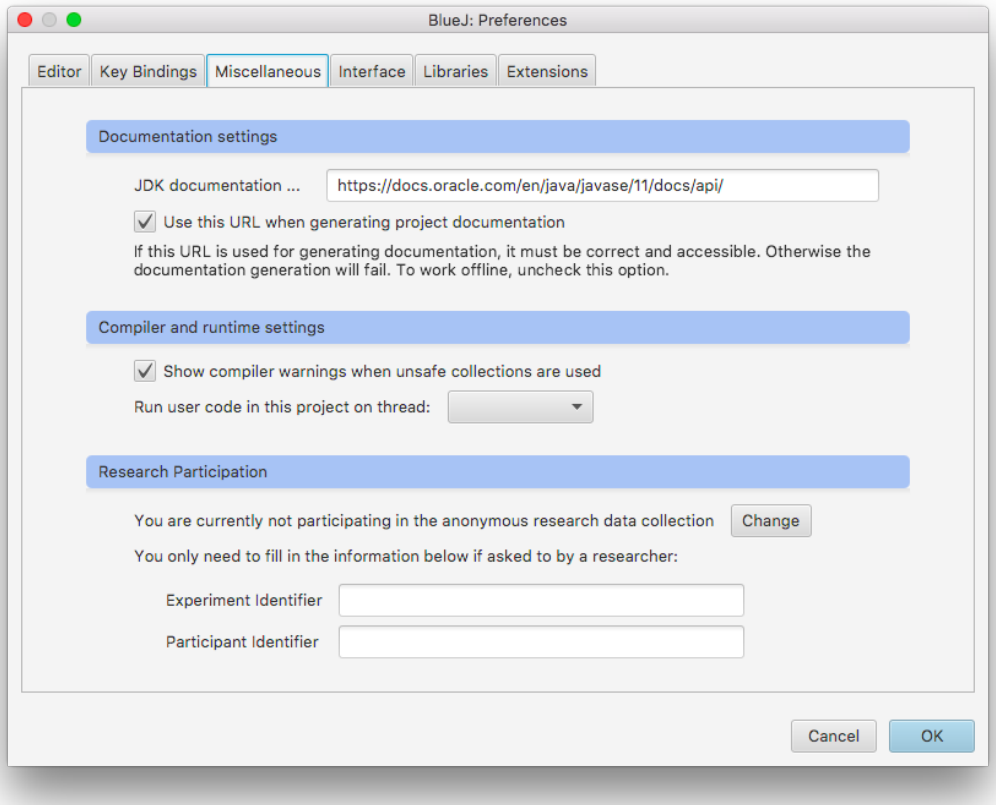
Let me get into more detail about which preferences I use and why.

- I use syntax highlighting (colored keywords).
- I display my line numbers.
- I auto-indent.
- I let BlueJ match brackets for me.
- I set the scope highlighting as dark as possible.

I think the meaning of most of those is clear, except for *bracket matching*. That means if I put my cursor right after a bracket or parenthesis in the code, BlueJ will highlight the matching bracket for me in gray.

If you have questions about any other of these preferences, please share them with other students in the Discussion Area.

The only other preference I'll mention now is under the Miscellaneous tab. Here's what it looks like:



BlueJ Miscellaneous Preferences

You shouldn't have to change anything on this tab. The important thing, though, is the JDK documentation URL, which points to the location of the documentation for the Java version you're using (probably version 11 or greater for most of you). That documentation opens whenever you're in BlueJ and choose Help > Java Class Libraries. Looking at that documentation as a beginner can be intimidating. But don't worry; stick with it and eventually it will all make perfect sense.

You can close the Preferences window at any time be clicking its OK button.

Summary

Congratulations! You've made it through another lesson.

I know we've covered a lot of ground, and I don't expect you to remember everything the first time through. But believe me, by the time this course is done, you'll be able to breeze through a lot of this. So take a deep breath, let it out, and relax.

In our next lesson, we'll rewrite HelloWorld in a more object-oriented programming (OOP) style.