The Java™ Tutorials

Trail: Getting Started

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Lesson: A Closer Look at the "Hello World!" Application

Now that you've seen the "Hello World!" application (and perhaps even compiled and run it), you might be wondering how it works. Here again is its code:

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

The "Hello World!" application consists of three primary components: source code comments, the `HelloWorldApp` class definition, and the `main` method. The following explanation will provide you with a basic understanding of the code, but the deeper implications will only become apparent after you've finished reading the rest of the tutorial.

Source Code Comments

The following bold text defines the *comments* of the "Hello World!" application:

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

Comments are ignored by the compiler but are useful to other programmers. The Java programming language supports three kinds of comments:

`/* text */`
: The compiler ignores everything from `/*` to `*/`.

`/** documentation */`
: This indicates a documentation comment (*doc comment*, for short). The compiler ignores this kind of comment, just like it ignores comments that use `/*` and `*/`. The `javadoc` tool uses doc comments when preparing automatically generated documentation. For more information on `javadoc`, see the Javadoc™ tool documentation .

`// text`
: The compiler ignores everything from `//` to the end of the line.

The `HelloWorldApp` Class Definition

The following bold text begins the class definition block for the "Hello World!" application:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // Display the string.
    }
}
```

As shown above, the most basic form of a class definition is:

```
class name {
    . . .
}
```

The keyword `class` begins the class definition for a class named `name`, and the code for each class appears between the opening and closing curly braces marked in bold above. Chapter 2 provides an overview of classes in general, and Chapter 4 discusses classes in detail. For now it is enough to know that every application begins with a class definition.

The `main` Method

The following bold text begins the definition of the `main` method:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

In the Java programming language, every application must contain a `main` method whose signature is:

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

The modifiers `public` and `static` can be written in either order (`public static` or `static public`), but the convention is to use `public static` as shown above. You can name the argument anything you want, but most programmers choose "args" or "argv".

The `main` method is similar to the `main` function in C and C++; it's the entry point for your application and will subsequently invoke all the other methods required by your program.

The `main` method accepts a single argument: an array of elements of type `String`.

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

This array is the mechanism through which the runtime system passes information to your application. For example:

```
java MyApp arg1 arg2
```

Each string in the array is called a *command-line argument*. Command-line arguments let users affect the operation of the application without recompiling it. For example, a sorting program might allow the user to specify that the data be sorted in descending order with this command-line argument:

```
-descending
```

The "Hello World!" application ignores its command-line arguments, but you should be aware of the fact that such arguments do exist.

Finally, the line:

```
System.out.println("Hello World!");
```

uses the `System` class from the core library to print the "Hello World!" message to standard output. Portions of this library (also known as the "Application Programming Interface", or "API") will be discussed throughout the remainder of the tutorial.

Previous page: Previous Lesson
Next page: Questions and Exercises: Getting Started