2. Creating Your First Java App

Contents

[1. Introduction 1](#_Toc7857)

[2. Creating Your App with Spring Tool Suite 1](#_Toc2005)

[3. IDE, JDK, and JRE 1](#_Toc30244)

[4. Running from the Command Line 1](#_Toc28402)

[5. Locating the Java Command 2](#_Toc13952)

[6. Statement Structure and Whitespace 2](#_Toc4706)

[7. Comments 2](#_Toc19919)

[8. Using Comments in Code 2](#_Toc14913)

[9. Introducing Packages 2](#_Toc13549)

[10. Using Packages 2](#_Toc28748)

[11. Summary 2](#_Toc14773)

# 1. Introduction

=>slides: Pg. 1

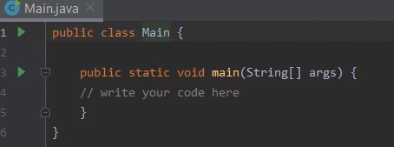
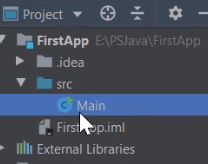
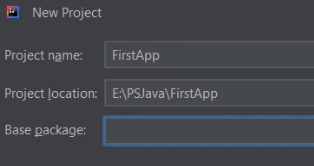
Welcome to the course, Getting Started with Programming in Java. In this module, we'll look at what's involved in creating a job application.

=>slides: Pg. 2

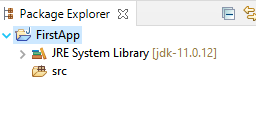
As we go through this module, we're going to start building up the fundamental skills that we need to work effectively as job application developers. So the first thing we're going to do is create and run an application as a developer does that. We're going to use a tool called Spring Tool Suite. Now, once we do that, we want to see what it's like for an end user to run her application. We'll see how we can run the application directly from the command line without having to have an entire development environment. From there, we're going to look at the basic statement syntax of Java, so we'll see how statements are structured within the Java programming language. After that, we're going to look at comments. Comments allow us to incorporate text into our application that's not processed by the compiler, and there are two fundamental types of comments, line comments and blocked comments. Then we're going to finish up. We'll look at packages. Now packages are a really important concept. We're going to talk about packages in much more detail later in the course series, but there's a few fundamental things we need to understand about packages in order to get us started, and that's we're going to focus on in this module.

# Creating Your App with Spring Tool Suite

Setup and Installation ppt



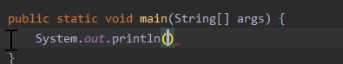
So now to kick off the course, the first thing we'll do is build a really simple Java application, and the reason we do that is that I've found whenever I'm working with a new language or a new environment, it's really helpful to build a simple application and run it just so I can see how everything works. Now, the tool we'll be using throughout this course and throughout this series is something called Spring Tool Suite, or more commonly known as, simply, STS, and it's one the most popular environments for creating Java and Spring applications. So now to create our application, we'll need to create a new project. We have this option here, File => New => Java Project. Again, choose that.



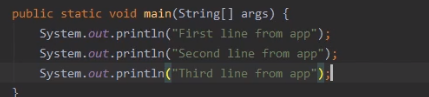
So now the next thing I need to do is name the application, and I'll name it FirstApp. Then once we name the project, we need to specify its location. You can see there, the STS is specifying a location to place my project. I like that location, so I'll just go ahead and stay with that.

So now with that, we're all set to create our project. So I'll head down here to the Finish button. I'll go and click that. Now, once our project is created, we can see we have source code here on the right side of the screen. On the left side, here, we have what's called the project view, or the project window, that simply shows the contents inside the project. Now, the first node here is FirstApp, the name of our project. We're going to expand that, and there's a few folders under there, but the one we're interested in is this guy, src, our source folder. I'll expand that.

We create a file called Main under the ‘src’ folder. This Main file is the source code file that we're currently looking at. If you look at that file, you see that it contains a class named Main. We'll talk more about classes a little later in the series. For now, you can just think of a class as something that can contain sections of code. When we create an application, the name of the class doesn't matter, but what does matter is the name of the method within the class because we have a method here named Main and the entry point for an application always has to be a method named Main and it has to have the same modifiers and the same look that this method does. So it has to be in public static void. As we go along, we'll talk about what all those things mean. Within the parentheses, we have to have the same content we have here with Main. So that's a specific style of entry point that the Java environment's going to look for when your applications run. We'll talk much more about all these things as we go along. So let's go and add a little bit of code to our application. We have this line here that says, write your code here. Let me start out by removing that.



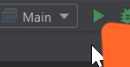
So now with that gone, let's add some of our own code. So we'll start out by saying, System.out.println. That System.out.println is simply a mechanism to display information to the screen. The information I want to print out will be a quoted string. We'll say, First line from app. Then after that, I'll add a semicolon. So what that will do now is print out the text, First line from app, when I run the application. So let's go down two more lines here, one that says, Second line from app and one that says, Third line from app. So now, with that, we're ready to build and run our application.



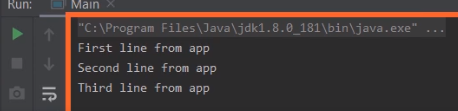
Now, fortunately, STS manages that process for us. We can simply tell it that we want the application to run, and it'll take care of the details of building it first. Now there's a couple of different ways we can run it.



One way, head up to our menus up here where it says, Run. I'm going to expand that. Then down under Run, you see there's an option that says, Run Main, that has a green arrow next to it. So if I selected that, it would build my application and go ahead and run it,



but you'll notice, if I go and collapse this menu, on the toolbar off to the right here, we have that same green arrow. Well, if I click on that green arrow, it will do the same thing, build and run the application, so let me go ahead and select that.



And then once I did that, STS took care of building the application for me, launched it, and it shows me the output in this window down here at the bottom. So I could see the output from our application, First line from app, Second line from app, Third line from app.. So that shows us that we've successfully created, built, and run our first Java application. So in our next effort, let's take a closer look at the parts of our Java environment.

# IDE, JDK, and JRE

=>slides: Pg. 3

When you first start working with Java, you'll find you'll run into a lot of acronyms, and all these acronyms can sometimes be confusing, so let's take a look at some of the key acronyms involved in developing Java applications. Now one of those acronyms is the JRE. The JRE simply stands for the Java Runtime Environment. The Java Runtime Environment is what enables our Java code to execute. The issue with Java is that because Java is a cross‑platform environment, when you build your Java application, it doesn't contain the native code that can run in a host environment. It contains what are called byte codes, and those byte codes have to be able to be translated somehow and able to execute within a particular host environment. It's the JRE that makes that execution possible. Then we have the JDK. The JDK is the Java Development Kit, and it's the Java Development Kit that provides the tools that are required to create our Java applications. Now, strictly speaking, the JDK is something that's provided by Oracle Corporation, but there's also an open source version known as the OpenJDK. Fundamentally, they behave very similarly. The key difference is the licensing. The JDK provided by Oracle is licensed under Oracle, where the OpenJDK is an open source license. Now as developers, one of the most important acronyms to us is the IDE or the IDE is our integrated development environment, and throughout this course series, the IDE we're using is IntelliJ. And it's the IDE that we interact with to edit, build, run, and debug our applications. But the IDE does not stand‑alone. It has to cooperate with the JDK and the JRE. So let's see what our setup looks like for us as developers.

=>slides: Pg. 4

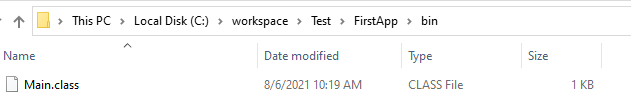
So we're going to start out again with our IDE, and that's where we editor application code, as well as initiate the builds and runs. But the IDE is not doing all that work on its own. When a build has to be done, the IDE cooperates with the JDK. So when we're in our IDE and we launch our build. It's actually passing that work off to the JDK, and it's the JDK that actually produces our Java application. Now when we produce a Java application, we want to run it in some host environment. Again, that might be our Windows operating system or Linux operating system, Mac operating system, or some other host environment, and that Java application can't run directly in that host environment because it's platform agnostic. All right, it contains these things called byte codes. So it's the JRE that allows the application to run. So although our developer experience may appear to be happening entirely within the IDE, it's cooperating with the JDK to do the builds, and the JRE to execute our application code.

=>slides: Pg. 4

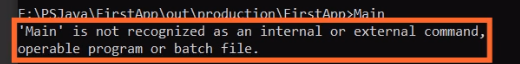
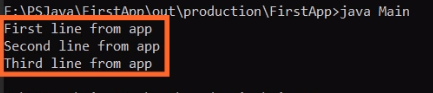
Now you may be thinking, well, that's a pretty complex setup environment. What about users who want to run my applications?   
=>slides: Pg. 5

Well, for the user, the set up is much, much simpler. Now, of course, a user is going to start out with some host environment. Again, it might be Windows, Mac, Linux or something else. Well, in the user's host environment, all we need to do is have the JRE installed. Once that JRE is installed on the host environment, Java applications become just another application to the user. The user can simply install that application, and as far as they're concerned, they run it just like any other application. So now in our next clip, let's see what it's like to run the application we just created as a user might run it.

# Running from the Command Line



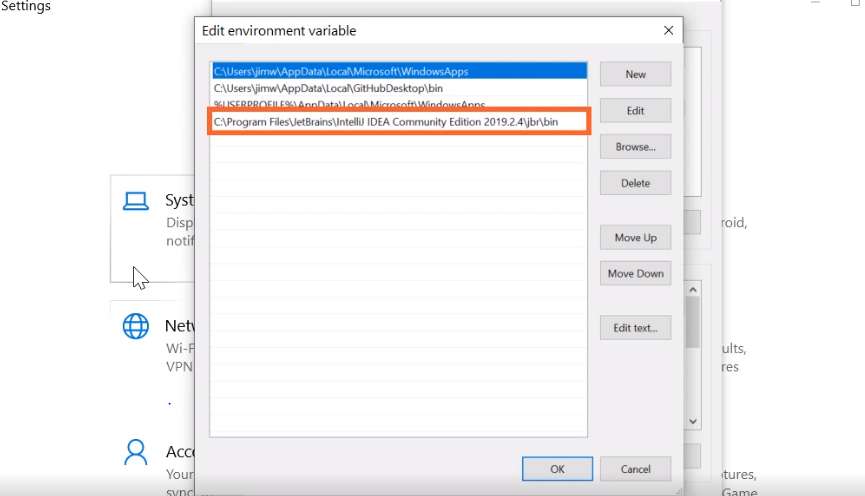
Earlier in this module, when we ran our application, we ran it from directly inside of IntelliJ. In other words, we ran it from directly inside of our IDE. Well, end users won't have access to the IDE. So the question is, how does an end user run our application? Well, the first thing they'll need is a copy of the application itself, and that's this file here, Main.class. Remember that when we created our application, it contained a class file named Main. Well, when we built this application, the compiler created this file, Main.class, So Main.class is a compiled version of our class Main. Now, to find that file, you'll first look in your project folder. So remember,. in my case, my project folder was (E:) PSJava, and then FirstApp. Then when we did the build, IntelliJ created, some additional folders, created an out folder, then the production folder, then another folder named after the project. Now, since our application is a command line application, we'll need to run it from the command line. Now a shortcut that we can use to launch a command line that's already in this folder is head up here to this box at the top. I'll go ahead and click in that. I'll type cmd. I'll hit Enter. So now the command prompt, that's already located at the correct folder.   
To confirm that's the correct folder, let's do a directory listing. So I'll type dir. I'll hit Enter, and when I do that, you can see our Main.class file.

  
 Now since the Main.class file is a program, it might seem like we can simply type the program name and then run it that way, but notice when we do that, we get a message that says that Main is not a recognised command.   
  
The thing we have to remember is that Java applications are not native applications. They have to be run from within a Java environment. So to create that Java environment, we'll need to use the Java command. So the Java command will launch an application into that Java environment. So we'll specify our class name. Now something that's very important to remember, class file names are case sensitive. So now we have our Java command followed by our class file name, I'll go ahead and hit Enter, and now you can see that our application ran. So that shows us that what the end user needs in order to run our application is two things, a copy of the application, which is the class file, but they'll also need this Java command. So in our next clip, let's take a look at where this Java command came from.

# Locating the Java Command

As we saw in the previous clip, we use the Java command to launch our job applications, and it turns out that the Java command is, itself, an application. It's a native application that runs Java applications. As with most native applications, in order to use the Java app name as a command, you need to include the app's location in your path environment variable. And this is true whether you're using Mac, Linux, or Windows. All three platforms support a path environment variable. So let's take a quick look at how to set the path environment variable here in Windows.



So we'll first switch to file explorer, and here, in this file explorer window, you can see java.exe. This is the Java command. Now the exact location of the command will depend on which version of the Java environment you've installed onto your system. The one we're looking at here is the Java environment that's automatically installed when you install IntelliJ. To use Java as a command, as I did in the previous clip, I had to have this folder, C: Program Files \? the name of the java edition jdk/ bin, included in my path environment variable.  
  
 So to access our environment variables, I'll press the Windows key and the letter I. That opens up our system settings. Now we need to view advanced system settings, so I'll start typing that. I'll then select view advanced system settings. I'll then choose environment variables. I'll head up here and I'll choose Path, head over here to where it says Edit. Let me go and expand this window a little bit, and as you can see, on my system I already have the location of a Java command listed here in my path environment variable. That's why I was able to type Java without having to fully qualify its location. To add the location to your path environment. variable, you can simply click the Browse button and then navigate to the appropriate folder. Once that's complete, you'll be able to use Java as a command in any new command prompts that you open. Alright, in our next clip, let's take a look at how Java statements are structured.

# Statement Structure and Whitespace

=>slides: Pg. 6

As we build our applications in Java, we're going to do that using a series of statements. Now there are a number of different types of statements in Java, and we'll look at the various types of statements as we go throughout this course and throughout this course series, but fundamentally, programs are made up of statements. So in our earlier application we wrote content out. We did that using a statement. So this is a statement that prints out the word Hello. Now there are few things to understand about how statements are structured. First of all, statements always end with a semicolon. Java is a language where the newline character, in general, doesn't really have any special meaning. The way we indicate that a statement is ending is by using that semicolon. Now, statements can contain zero or more white spaces. White spaces simply serve as visual separators. So this statement we have here, we have a single space after the opening parenthesis and a single space before the closing parenthesis. Now white spaces aren't limited to the space character. White spaces include the space character, the tab character and the newline character, in other words, the Enter key is just the white space. So this nature of white space gives us a lot of flexibility in how we structure our statements.

=>slides: Pg. 7

So we could write a statement like this, where we have a few single spaces in it, or we can write the same statement with no spaces, or the same statement with a mixture of spaces throughout it, or we could even write it like this, where there's a mixture of spaces and newlines mixed throughout it. No matter how you format the statement with white space, there's no change in the statement's behavior. It's a statement content that controls behavior, white spaces have no effect. All right,, so now in our next clip, let's take a look at comments.

# Comments

=>slides: Pg. 8

Java allows us to place comments within our source code, and a comment is simply a string of text that's ignored by the compiler, and there's a few common reasons we use comments. One is to simply place human readable notes within our source code. It just allows us to annotate within the source code the work that we're trying to do. It could be useful to me as the person creating the code so when I come back later I can remember what I was doing. It could be really useful to someone has to maintain the code. Now, also, we sometimes use comments to hide source code without deleting it. Maybe we have one or more statements that we don't want to have run right now, but we don't want to delete them. Well, in that case, we can use a comment to prevent the compiler from processing that statement.

=>slides: Pg. 9  
Java supports three general types of comments. One is what we call the line comment, and that's indicated by the // characters. And the line comment tells the compiler to ignore the text until it reaches the end of the current line. Now, this is a case where the new line character actually matters. The compiler will ignore everything after the // until it reaches a new line character. Now, another type of comment is what we call the block comment. A block comment begins with a /\* and continues until the \*/ is reached and all text within that block is ignored. And this allows us to have comments that start and end within a particular line or even span multiple lines. And there's another type of comment known as a Javadoc comment. The Javadoc comment begins with /\*\* and ends when it reaches the \*/. Now, from the compiler standpoint, a Javadoc comment is treated just like a block comment, so all the text between the opening and closing characters are ignored by the compiler. But the key thing is that Javadoc comments could be used to generate documentation. Now the details of Javadoc are outside the scope of this course, but fundamentally, what it does is provide a syntax that allows us to write documentation for our code directly in the source code. So there's a utility called Javadoc that can generate that documentation. In addition, some IDE's, including IntelliJ, can dynamically generate that documentation using these Javadoc comments. Okay, so now in our next clip, let's see how we can use line comments and block comments within our application code.

# Using Comments in Code



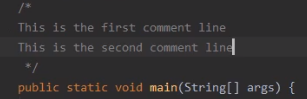
Here we are back in STS. And what want to do now is see how adding comments affects our application code. So let's start out by looking at line comments. So we have this line here where we write our first line of output. Let's add a line comment to the end of that line. So remember that we indicated our line comment by using //. And as a line comment, that // indicates that everything is to be ignored until we reach the end of the current line, in other words until we reach the new line character. And you can see there that the text This is a line comment appears in a different color than the rest of our source code, indicating that it's a comment.



But remember that as a comment, it doesn't affect our application behavior. So if I go over and I run my application, you can see the output of the application is exactly as it was before I added the comment because, again, the comment is ignored by the compiler. But now because the comment is ignored by the compiler, we can actually use comments to comment out existing code. So we have this line here where we write out Second line from app. If I don't want that code to run, I can simply put the // at the beginning of that line. And you notice that once I do that, that entire line appears in the same color as the other line comment I had because as far as the compiler is concerned, that's no longer an executable line of code. It's just a comment. So if I run the program again, and notice that our output only says First line from app and Third line from app because the line that said Second line from app no longer runs. Now, of course, we can use line comments on a line by themselves.



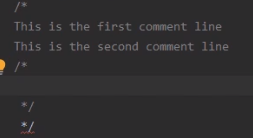
So up here, let's go ahead and add a standalone line comment. And again, as a comment, it has no impact on the execution of our application. It simply gives us the opportunity to include some additional text within our application. All right, so now let's take a look at block comments.



So let's go up here, and we'll start a block comment by saying /\*. So we have that /\* to indicate the start of a block comment. Now one thing to keep in mind, when we say /\* or /\*, we're talking about the exact same characters. They're just two different ways of saying the same syntax. It's the syntax we use to start a block comment. Then, once we have that block comment started, of course, we want to go ahead and end it so we do that with an \*/. Now remember as a block comment, we can actually have multiple lines of text within here. So now we have that multi‑line comment there. Lets go ahead and run our application. And once we run it, just as expected, our output is unchanged because the comment text doesn't affect the execution of our program. Now one of the cool things about block comments is they have a distinct starting and a distinct ending point. Well because they have distinct starting and ending points, we can use them not only for multi‑line comments, but we can also use them for comments within the middle of a line.



So look down here where we right out the third line from app. Well, let's say we want to temporarily change that text that we're writing out. Well we can take that text string and enclose it within a block comment. So that text for Third line from app will now be ignored by the compiler. And instead, we might write out some different text. So now, let's go ahead and run our application. Instead of writing out Third line from app, it writes out This is different text because, again, the block comment allowed our comment to have a distinct starting point, the /\*, and a distinct ending point, the \*/.



So now let's take a look at nesting comments within a block comment. So we have our block comment up top here. So let's go ahead and start another block comment within this block comment. So we'll start out with /\*. Then we'll go ahead and add our \*/. And notice as soon as I do that, the second \*/ is underlined in red, and that's indicating that we now have an error. And the issue is that you cannot nest one block comment inside of another. The way block comments work is that once the compiler sees the /\*, it knows that a block comment is starting. So it ignores everything until it sees the first \*/. As soon as it sees that \*/, it no longer considers itself to be part of a comment. So when it sees the second \*/, the compiler is simply saying, I don't know what that is. So whenever you use a block comment, we can't nest a second block comment within it. So let's go ahead and remove that inner block comment. Now with that gone, we can nest a line comment within a block comment.



And you notice when I do that, there is no error because again, remember, the way that block comments work is that once it sees the /\*, it ignores everything until it sees the \*/. So when it encounters the nested line comment, that doesn't create a problem. All right so now in our next clip, let's take a quick look at Java packages.

# Introducing Packages

=>slides: Pg. 10

So let's take a look now at a concept in Java known as packages, and packages have a lot of important capabilities. We're going to talk about packages in detail later in this course series. There's just a few key things we want to understand to get us started. Now remember, when we created the application earlier in this module, we didn't use a package name. So our application source file looked something like this. It was simply a class Main and then the code within that class. Well, normally, that's not how we create source files. We normally qualify our source files with a package name. We do that by using the package keyword followed by the name we want to give that package. And one of the key things packages do is provide organization. Now they also follow a standard naming convention, and as we'll see, that naming convention assures that our package names are unique. And also, package names affect where our source code is structured. So let's start out by looking at the naming conventions for packages.

=>slides: Pg. 11

Now by convention, package names are all lowercase, and they use something known as reverse domain name notation. And this notation assures global uniqueness of our package names. So here at mycompany, when we name our packages, we might follow a naming convention like this, com.mycompany. Now that would probably look pretty familiar. If you were to go to your web browser and you wanted to go to mycompany website, you would go to mycompany.com because mycompany.com is a domain name owned by mycompany. And there's actually an authority that makes sure that no one else can buy that name. So what we do with our package names is we simply reverse that. Instead of using mycompany.com, we use com.mycompany. And so that's what we mean by using reverse domain name notation. From there, we further qualify our package names to assure that they're unique within the organization. So, for example, all the code related to a search within mycompany might be contained in a package named com.mycompany.search. So the .search indicates that it's code related to search. But as an organization gets larger, we may need more detailed organization. We may want to add information related to the group within the company that's creating the code. Say, for example, that the mycompany sales organization had its own group of developers. Well when they name packages, they would start out with com.mycompany. But what they might do is at a .sales onto that to indicate that it's being created by the developer team within the sales organization. So if they were creating an application that managed sales accounts, they might name it com.mycompany.sales.accountmanagement. And that's really helpful because even if there was another group within .mycompany that wanted to create an account management application, you wouldn't have to worry about the package names colliding because they've included the group name of sales within the package name. Now making sure that sales group name is unique is the responsibility of mycompany. But because the com.mycompany already assures that everything within mycompanyis unique, as long as mycompanymanages that, everything will work out perfectly. So that way, if the Human Resource department also has their own team of developers, they might start out their package names as com.mycompany.humanresources. So if they created an application to manage user accounts of all employees within mycompany, they could safely name the application accountmanagement, and it wouldn't collide because we've already made it unique with com.mycompany.humanresources. By our package names being unique, it helps make our type names unique because type names are qualified by their package.

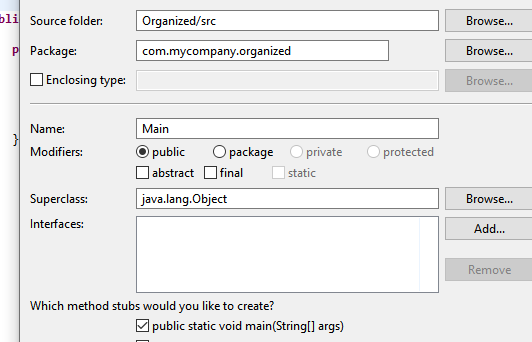
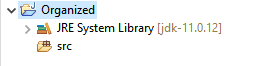
=>slides: Pg. 12

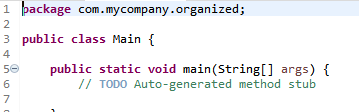
So that code we wrote earlier where we had a class named Main that was not part of a package, well, the name of that type would simply be Main. But by placing it in a package, that now qualifies the name within the package. So by putting our class Main inside the package name com.mycompany.example, the name of the type is now com.mycompany.example.main. So because our package names are globally unique, it now allows our type names to be globally unique. Now another important factor of package names is they affect our source code organization.

=>slides: Pg. 13

So if we were to look at the organization's source code under our project folder, we would find that underneath that project folder, there's a folder named src, and that's where all of our source code is located. And then underneath the src folder, there's a series of subfolders based on the package name. So with a package name of com.mycompany.example, under the src folder, there would be a com folder. Under the com folder, there will be a mycompany folder. Under the mycompany folder, there would be an example folder. And then the Main.java file that contains our Main class would be within that example folder. So as you can see, package names help assure uniqueness of our type names and give us a predictable organization of our source code. So now in our next clip, let's create another project, but this time, we'll use a package name.

# Using Packages

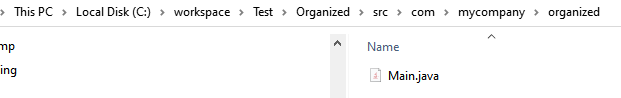




**public** **static** **void** main(String[] args) {

System.***out***.println("We got organized!");

}



# Summary

=>slides: Pg. 14

To wrap up, here are some of the key things you want to remember from this module. Remember that as developers, we tend to work with what's called an integrated development environment or also known as an IDE. This is what provides our development experience. So it's the IDE that we're going to use to edit our code, to initiate our builds, as well as run and debug or applications. But now remember, although the IDE provides a developer experience, it doesn't work alone. In order to do our build, we rely on something known as the Java Development Kit or the JDK. And that has the actual tools in it that we need to create our Java applications. So when we initiate a build within the IDE, it's actually cooperating with the JDK to do the work of building the application. Now remember that technically, JDK is licensed by a particular corporation, but we also have an open source version, which is OpenJDK. So that gives us access to the ability to build our Java applications while using an open source license as opposed to a license provided by a particular corporation. And then to run our applications, we rely on something known as the Java Runtime Environment. Remember that when we build our Java code, the application that's produced is platform‑independent. So it doesn't target a particular operating system. So it needs something to enable the Java code execution, and that's what the JRE does. It produces the Java environment on a particular host machine, which allows our Java code to run.

=>slides: Pg. 15

Now remember, as we build up our applications, we're going to do that using a series of statements. And although each statement type has a particular structure, there's a few things that they have in common. One thing is that statements end with a semicolon. So we rely on that semicolon to indicate when a particular statement ends. Then the parts of a statement are separated by zero or more whitespaces, and whitespaces include the space character, the tab character, and the new line. So placing a new line within a statement doesn't have any particular meaning. That's why we rely on the semicolon to indicate when a statement ends. Now remember, we also have comments, and comments allows to have text in our source code that's not processed by the compiler. And we use that for two main reasons. One is to allow human‑readable notes to be placed within the source code. But also, it allows us to hide certain statements from the compiler that we don't want to have currently run, but we don't want to delete them. We can use comments to hide that code so they're not part of the application when we do our build. Then we finished up with a look at packages. And as we mentioned, packages do a number of important things. But the ones we focused on in this module were the idea of type uniqueness. Remember the convention we use for naming packages assures that each package name is unique. And because type names are qualified by their packages, packages assure that each type name is also unique. And also, packages affect our source code file structure. Remember that each part of the package name becomes a subfolder within our source code folder, and that provides organization of the way our source code is managed. All right, that wraps up this module. In our next module, we'll start taking a look at Java data types, how we declare variables, and how we use Java math operators.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*