Chapter 1
Welcome to Java

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Understanding Java Technology and Environment**
  - Describe Java Technology and the Java development environment
  - Identify key features of the Java language
- **Creating a Simple Java Program**
  - Create an executable Java program with a main class
  - Compile and run a Java program from the command line
  - Create and import packages
- **Describing and Using Objects and Classes**
  - Define the structure of a Java class

Welcome to the beginning of your journey to achieve a Java 11 certification. We assume this isn't the first Java programming book you've read. Although we do talk about the basics, we do so only because we want to make sure you have all the terminology and detail you'll need for the 1Z0-815 exam. If you've never written a Java program before, we recommend you pick up an introductory book on any version of Java. Examples include *Head First Java*, *2nd Edition* (O'Reilly Media, 2009); *Java for Dummies* (For Dummies, 2017), *Murach's Java Programming* (Murach, 2017), or *Thinking in Java, 4th Edition* (Prentice Hall, 2006). It's okay if the book covers an older version of Java—even Java 1.3 is fine. Then come back to this certification study guide.

This chapter covers the fundamentals of Java. You'll better understand the Java environments and benefits of Java. You'll also see how to define and run a Java class and learn about packages.

Learning About the Java Environment

The Java environment consists of understanding a number of technologies. In the following sections, we will go over the key terms and acronyms you need to know for the exam and then discuss what software you need to study for the exam.

Major Components of Java

The *Java Development Kit* (JDK) contains the minimum software you need to do Java development. Key pieces include the compiler (`javac`), which converts `.java` files to `.class` files, and the launcher `java`, which creates the virtual machine and executes the program. We will use both later in this chapter when running programs at the command line. The JDK also contains other tools including the archiver (`jar`) command, which can package files together, and the API documentation (`javadoc`) command for generating documentation.

The `javac` program generates instructions in a special format that the `java` command can run called *bytecode*. Then `java` launches the *Java Virtual Machine* (JVM) before running the code. The JVM knows how to run bytecode on the actual machine it is on. You can think of the JVM as a special magic box on your machine that knows how to run your `.class` file.

WHERE DID THE JRE GO?

In previous versions of Java, you could download a Java Runtime Environment (JRE) instead of the full JDK. The JRE was a subset of the JDK that was used for running a program but could not compile one. It was literally a subset. In fact, if you looked inside the directory structure of a JDK in older versions of Java, you would see a folder named `jre`.

In Java 11, the JRE is no longer available as a stand-alone download or a subdirectory of the JDK. People can use the full JDK when running a Java program. Alternatively, developers can supply an executable that contains the required pieces that would have been in the JRE. The `jlink` command creates this executable.

While the JRE is not in scope for the exam, knowing what changed may help you eliminate wrong answers.

When writing a program, there are common pieces of functionality and algorithms that developers need. Luckily, we do not have to write each of these ourselves. Java comes with a large suite of *application programming interfaces* (APIs) that you can use. For example, there is a `StringBuilder` class to create a large `String` and a method in `Collections` to sort a list. When writing a program, it is helpful to look what pieces of your assignment can be accomplished by existing APIs.

You might have noticed that we said the JDK contains the minimum software you need. Many developers use an *integrated development environment* (IDE) to make writing and running code easier. While we do not recommend using one while studying for the exam, it is still good to know that they exist. Common Java IDEs include Eclipse, IntelliJ IDEA, and NetBeans.

Downloading a JDK

Every six months, the version number of Java gets incremented. Java 11 came out in September 2018. This means that Java 11 will not be the latest version when you download the JDK to study for the exam. However, you should still use Java 11 to study with since this is a Java 11 exam. The rules and behavior can change with later versions of Java. You wouldn't want to get a question wrong because you studied with a different version of Java!

Every three years, Oracle has a *long-term support* (LTS) release. Unlike non-LTS versions that are supported for only six months, LTS releases have patches and upgrades available for at least three years. Even after the next LTS, Java 17, comes out, be sure to use Java 11 to study for the Java 11 certification exam.

NOTE

Oracle changed the licensing model for its JDK. While this isn't on the exam, you can read more about the licensing changes and other JDKs from the links on our book's website:

http://www.selikoff.net/ocp11-complete/

We recommend using the Oracle distribution of Java 11 to study for this exam. Note that Oracle's JDK is free for personal use as well as other scenarios. Alternatively, you can use OpenJDK, which is based on the same source code.

NOTE

The Oracle distribution requires you to register for an Oracle account if you don't already have one. This is the same Oracle account you will use to get your exam scores, so you will have to do this at some point anyway.

Identifying Benefits of Java

Java has some key benefits that you'll need to know for the exam.

Object Oriented Java is an object-oriented language, which means all code is defined in classes, and most of those classes can be instantiated into objects. We'll discuss this more throughout the book. Many languages before Java were procedural, which meant there were routines or methods but no classes. Another common approach is functional programming. Java allows for functional programming within a class, but object-oriented is still the main organization of code.

Encapsulation Java supports access modifiers to protect data from unintended access and modification. Most people consider encapsulation to be an aspect of object-oriented languages. Since the exam objectives call attention to it specifically, so do we. In fact, Chapter 7, “Methods and Encapsulation,” covers it extensively.

Platform Independent Java is an interpreted language that gets compiled to bytecode. A key benefit is that Java code gets compiled once rather than needing to be recompiled for different operating systems. This is known as “write once, run everywhere.” The portability allows you to easily share pre-compiled pieces of software. When studying for the 1Z0-816 exam, you’ll learn that it is possible to write code that throws an exception in some environments, but not others. For example, you might refer to a file in a specific directory. If you get asked about running Java on different operating systems on the 1Z0-815 exam, the answer is that the same class files run everywhere.

Robust One of the major advantages of Java over C++ is that it prevents memory leaks. Java manages memory on its own and does garbage collection automatically. Bad memory management in C++ is a big source of errors in programs.

Simple Java was intended to be simpler to understand than C++. In addition to eliminating pointers, it got rid of operator overloading. In C++, you could write `a + b` and have it mean almost anything.

Secure Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on. On the 1Z0-816 exam, there is even an exam objective for security.

Multithreaded Java is designed to allow multiple pieces of code to run at the same time. There are also many APIs to facilitate this task. You’ll learn about some of them when studying for the 1Z0-816 exam.

Backward Compatibility The Java language architects pay careful attention to making sure old programs will work with later versions of Java. While this doesn’t always occur, changes that will break backward compatibility occur slowly and with notice. *Deprecation* is a technique to accomplish this where code is flagged to indicate it shouldn’t be used. This lets developers know a different approach is preferred so they can start changing the code.

Understanding the Java Class Structure

In Java programs, classes are the basic building blocks. When defining a *class*, you describe all the parts and characteristics of one of those building blocks. To use most classes, you have to create objects. An *object* is a runtime instance of a class in memory. An object is often referred to as an *instance* since it represents a single representation of the class. All the various objects of all the different classes represent the state of your program. A *reference* is a variable that points to an object.

In the following sections, we'll look at fields, methods, and comments. We'll also explore the relationship between classes and files.

Fields and Methods

Java classes have two primary elements: *methods*, often called functions or procedures in other languages, and *fields*, more generally known as variables. Together these are called the *members* of the class. Variables hold the state of the program, and methods operate on that state. If the change is important to remember, a variable stores that change. That's all classes really do. It's the programmer who creates and arranges these elements in such a way that the resulting code is useful and, ideally, easy for other programmers to understand.

Other building blocks include interfaces, which you'll learn about in Chapter 9, "Advanced Class Design," and enums, which you'll learn about in detail when you study for the 1Z0-816 exam.

The simplest Java class you can write looks like this:

```
1: public class Animal {
2: }
```

Java calls a word with special meaning a *keyword*. Other classes can use this class since there is a `public` keyword on line 1. The `class` keyword indicates you're defining a class. `Animal` gives the name of the class. Granted, this isn't an interesting class, so let's add your first field.

```
1: public class Animal {
2:    String name;
3: }
```

The line numbers aren't part of the program; they're just there to make the code easier to talk about.

On line 2, we define a variable named `name`. We also define the type of that variable to be a `String`. A `String` is a value that we can put text into, such as `"this is a string"`. `String` is also a class supplied with Java. Next you can add methods.

```
1: public class Animal {
2:    String name;
3:    public String getName() {
4:       return name;
5:    }
6:    public void setName(String newName) {
7:       name = newName;
8:    }
9: }
```

On lines 3–5, you've defined your first method. A method is an operation that can be called. Again, `public` is used to signify that this method may be called from other classes. Next comes the return type—in this case, the method returns a `String`. On lines 6–8 is another method. This one has a special return type called *void*. The `void` keyword means that no value at all is returned. This method requires information be supplied to it from the calling method; this information is called a *parameter*. The `setName()` method has one parameter named `newName`, and it is of type `String`. This means the caller should pass in one `String` parameter and expect nothing to be returned.

Two pieces of the method are special. The method name and parameter types are called the *method signature*. In this example, can you identify the method name and parameters?

```
public int numberVisitors(int month)
```

The method name is `numberVisitors`. There's one parameter named `month`, which is of type `int`, which is a numeric type.

The *method declaration* consists of additional information such as the return type. In this example, the return type is `int`.

Comments

Another common part of the code is called a *comment*. Because comments aren't executable code, you can place them in many places. Comments can make your code easier to read. You won't see many comments on the exam since the exam creators are trying to make the code harder to read. You will see them in this book as we explain the code. And we hope you use them in your own code. There are three types of comments in Java. The first is called a single-line comment:

```
// comment until end of line
```

A single-line comment begins with two slashes. The compiler ignores anything you type after that on the same line. Next comes the multiple-line comment:

```
/* Multiple
 * line comment
 */
```

A multiple-line comment (also known as a multiline comment) includes anything starting from the symbol `/*` until the symbol `*/`. People often type an asterisk (`*`) at the beginning of each line of a multiline comment to make it easier to read, but you don't have to. Finally, we have a Javadoc comment:

```
/**
 * Javadoc multiple-line comment
 * @author Jeanne and Scott
 */
```

This comment is similar to a multiline comment except it starts with `/**`. This special syntax tells the Javadoc tool to pay attention to the comment. Javadoc comments have a specific structure that the Javadoc tool knows how to read. You probably won't see a Javadoc comment on the exam. Just remember it exists so you can read up on it online when you start writing programs for others to use.

As a bit of practice, can you identify which type of comment each of the following six words is in? Is it a single-line or a multiline comment?

```
/*
 * // anteater
 */
// bear
// // cat
// /* dog */
/* elephant */
/*
 * /* ferret */
 */
```

Did you look closely? Some of these are tricky. Even though comments technically aren't on the exam, it is good to practice to look at code carefully.

OK, on to the answers. The comment containing `anteater` is in a multiline comment. Everything between `/*` and `*/` is part of a multiline comment—even if it includes a single-line comment within it! The comment containing `bear` is your basic single-line comment. The comments containing `cat` and `dog` are also single-line comments. Everything from `//` to the end of the line is part of the comment, even if it is another type of comment. The comment containing `elephant` is your basic multiline comment.

The line with `ferret` is interesting in that it doesn't compile. Everything from the first `/*` to the first `*/` is part of the comment, which means the compiler sees something like this:

```
/* */ */
```

We have a problem. There is an extra `*/`. That's not valid syntax—a fact the compiler is happy to inform you about.

Classes vs. Files

Most of the time, each Java class is defined in its own `.java` file. It is usually `public`, which means any code can call it. Interestingly, Java does not require that the class be `public`. For example, this class is just fine:

```
class Animal {
   String name;
}
```

You can even put two classes in the same file. When you do so, at most one of the classes in the file is allowed to be public. That means a file containing the following is also fine:

```
public class Animal {
   private String name;
}
class Animal2 {
}
```

If you do have a public class, it needs to match the filename. The declaration `public class Animal2` would not compile in a file named `Animal.java`. In Chapter 7, we will discuss what access options are available other than `public`.

Writing a *main()* Method

A Java program begins execution with its `main()` method. A `main()` method is the gateway between the startup of a Java process, which is managed by the Java Virtual Machine (JVM), and the beginning of the programmer's code. The JVM calls on the underlying system to allocate memory and CPU time, access files, and so on. In this section, you will learn how to create a `main()` method, pass a parameter, and run a program both with and without the `javac` step.

CHECKING YOUR VERSION OF JAVA

Before we go any further, please take this opportunity to ensure you have the right version of Java on your path.

```
javac -version
java -version
```

Both of these commands should include a version number that begins with the number 11.

Creating a *main()* Method

The `main()` method lets the JVM call our code. The simplest possible class with a `main()` method looks like this:

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

   }
}
```

This code doesn't do anything useful (or harmful). It has no instructions other than to declare the entry point. It does illustrate, in a sense, that what you can put in a `main()` method is arbitrary. Any legal Java code will do. In fact, the only reason we even need a class structure to start a Java program is because the language requires it. To compile and execute this code, type it into a file called `Zoo.java` and execute the following:

```
javac Zoo.java
java Zoo
```

If you don't get any error messages, you were successful. If you do get error messages, check that you've installed the Java 11 JDK, that you have added it to the `PATH`, and that you didn't make any typos in the example. If you have any of these problems and don't know what to do, post a question with the error message you received in the Beginning Java forum at CodeRanch (www.coderanch.com/forums/f-33/java).

To compile Java code, the file must have the extension `.java`. The name of the file must match the name of the class. The result is a file of bytecode by the same name, but with a `.class` filename extension.

Remember that bytecode consists of instructions that the JVM knows how to execute. Notice that we must omit the `.class` extension to run `Zoo.java`.

The rules for what a Java code file contains, and in what order, are more detailed than what we have explained so far (there is more on this topic later in the chapter). To keep things simple for now, we'll follow this subset of the rules:

- Each file can contain only one public class.
- The filename must match the class name, including case, and have a `.java` extension.

Suppose we replace line 3 in `Zoo.java` with the following:

```
3: System.out.println("Welcome!");
```

When we compile and run the code again, we'll get the line of output that matches what's between the quotes. In other words, the program will output `Welcome!`.

Let's first review the words in the `main()` method's signature, one at a time. The keyword `public` is what's called an *access modifier*. It declares this method's level of exposure to potential callers in the program. Naturally, `public` means anyplace in the program. You'll learn more about access modifiers in Chapter 7.

The keyword *static* binds a method to its class so it can be called by just the class name, as in, for example, `Zoo.main()`. Java doesn't need to create an object to call the `main()` method—which is good since you haven't learned about creating objects yet! In fact, the JVM does this, more or less, when loading the class name given to it. If a `main()` method isn't present in the class we name with the `.java` executable, the process will throw an error and terminate. Even if a `main()` method is present, Java will throw an exception if it isn't static. A nonstatic `main()` method might as well be invisible from the point of view of the JVM. You'll see `static` again in Chapter 7.

The keyword `void` represents the *return type*. A method that returns no data returns control to the caller silently. In general, it's good practice to use `void` for methods that change an object's state. In that sense, the `main()` method changes the program state from started to finished. We will explore return types in Chapter 7 as well. (Are you excited for Chapter 7 yet?)

Finally, we arrive at the `main()` method's parameter list, represented as an array of `java.lang.String` objects. In practice, you can write any of the following:

```
String[] args
String args[]
String... args;
```

The compiler accepts any of these. The variable name `args` hints that this list contains values that were read in (arguments) when the JVM started. The characters `[]` are brackets and represent an array. An array is a fixed-size list of items that are all of the same type. The characters `...` are called varargs (variable argument lists). You will learn about `String` in Chapter 2, "Java Building Blocks." Arrays and varargs will follow in Chapter 5, "Core Java APIs."

While the previous example used the common `args` parameter name, you can use any valid variable name you like. The following three are also allowed:

```
String[] options
String options []
String... options;
```

Passing Parameters to a Java Program

Let's see how to send data to our program's `main()` method. First we modify the `Zoo` program to print out the first two arguments passed in:

```
public class Zoo {
   public static void main(String[] args) {
      System.out.println(args[0]);
      System.out.println(args[1]);
   }
}
```

The code `args[0]` accesses the first element of the array. That's right: array indexes begin with 0 in Java. To run it, type this:

```
javac Zoo.java
java Zoo Bronx Zoo
```

The output is what you might expect:

```
Bronx
Zoo
```

The program correctly identifies the first two "words" as the arguments. Spaces are used to separate the arguments. If you want spaces inside an argument, you need to use quotes as in this example:

```
javac Zoo.java
java Zoo "San Diego" Zoo
```

Now we have a space in the output:

```
San Diego
Zoo
```

To see if you follow that, what do you think this outputs?

```
javac Zoo.java
java Zoo San Diego Zoo
```

The answer is two lines. The first one is `San`, and the second is `Diego`. Since the program doesn't read from `args[2]`, the third element (`Zoo`) is ignored.

All command-line arguments are treated as `String` objects, even if they represent another data type like a number:

```
javac Zoo.java
java Zoo Zoo 2
```

No matter. You still get the values output as `String` values. In Chapter 2, you'll learn how to convert `String` values to numbers.

```
Zoo
2
```

Finally, what happens if you don't pass in enough arguments?

```
javac Zoo.java
java Zoo Zoo
```

Reading `args[0]` goes fine, and `Zoo` is printed out. Then Java panics. There's no second argument! What to do? Java prints out an exception telling you it has no idea what to do with this argument at position 1. (You'll learn about exceptions in Chapter 10, "Exceptions.")

```
Zoo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  Index 1 out of bounc
   at Zoo.main(Zoo.java:4)
```

To review, the JDK contains a compiler. Java class files run on the JVM and therefore run on any machine with Java rather than just the machine or operating system they happened to have been compiled on.

Running a Program in One Line

Starting in Java 11, you can run a program without compiling it first—well, without typing the `javac` command that is. Let's create a new class:

```
public class SingleFileZoo {
   public static void main(String[] args) {
      System.out.println("Single file: " + args[0]);
   }
}
```

We can run our `SingleFileZoo` example without actually having to compile it.

```
java SingleFileZoo.java Cleveland
```

Notice how this command passes the name of the Java file. When we compiled earlier, we wrote `java Zoo`. When running it as a one-liner, we write `java SingleFileZoo.java`. This is a key difference. After you first compiled with `javac`, you then passed the `java` command the name of the class. When running it directly, you pass the `java` command the name of the file. This feature is called launching *single-file source-code* programs. The name cleverly tells you that it can be used only if your program is one file. This means if your program has two `.java` files, you still need to use `javac`.

Now, suppose you have a class with invalid syntax in it. What do you think happens when we run `java Learning.java`?

```
public class Learning {
   public static void main(String[] args) {
      UhOh;  // DOES NOT COMPILE
      System.out.println("This works!");
   }
}
```

Java is still a compiled language, which means the code is being compiled in memory and the `java` command can give you a compiler error.

```
Learning.java:3: error: not a statement
   UhOh; // DOES NOT COMPILE
   ^
1 error
error: compilation failed
```

Notice how we said "in memory." Even if the code compiles properly, no `.class` file is created. This faster way of launching single-file source-code programs will save you time as you study for the exam. You'll be writing a lot of tiny programs. Having to write one line to run them instead of two will be a relief! However, compiling your code in advance using `javac` will result in the program running faster, and you will definitely want to do that for real programs.

Table 1.1 highlights the differences between this new feature and the traditional way of compiling. You'll learn about imports in the next section, but for now, just know they are a way of using code written by others.

TABLE 1.1 Running programs

Full command	**Single-file source-code command**

`javac HelloWorld.java`
`java HelloWorld`
	`java HelloWorld.java`

Produces a class file	Fully in memory
For any program	For programs with one file
Can import code in any available Java library	Can only import code that came with the JDK

Understanding Package Declarations and Imports

Java comes with thousands of built-in classes, and there are countless more from developers like you. With all those classes, Java needs a way to organize them. It handles this in a way similar to a file cabinet. You put all your pieces of paper in folders. Java puts classes in *packages*. These are logical groupings for classes.

We wouldn't put you in front of a file cabinet and tell you to find a specific paper. Instead, we'd tell you which folder to look in. Java works the same way. It needs you to tell it which packages to look in to find code.

Suppose you try to compile this code:

```
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();   // DOES NOT COMPILE
      System.out.println(r.nextInt(10));
   }
}
```

The Java compiler helpfully gives you an error that looks like this:

```
Random cannot be resolved to a type
```

This error could mean you made a typo in the name of the class. You double-check and discover that you didn't. The other cause of this error is omitting a needed *import* statement. Import statements tell Java which packages to look in for classes. Since you didn't tell Java where to look for `Random`, it has no clue.

Trying this again with the `import` allows you to compile.

```
import java.util.Random;  // import tells us where to find Random
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();
      System.out.println(r.nextInt(10));  // print a number 0-9
   }
}
```

Now the code runs; it prints out a random number between 0 and 9. Just like arrays, Java likes to begin counting with 0.

As you can see in the previous example, Java classes are grouped into packages. The `import` statement tells the compiler which package to look in to find a class. This is similar to how mailing a letter works. Imagine you are mailing a letter to 123 Main St., Apartment 9. The mail carrier first brings the letter to 123 Main St. Then she looks for the mailbox for apartment number 9. The address is like the package name in Java. The apartment number is like the class name in Java. Just as the mail carrier only looks at apartment numbers in the building, Java only looks for class names in the package.

Package names are hierarchical like the mail as well. The postal service starts with the top level, looking at your country first. You start reading a package name at the beginning too. If it begins with `java` or `javax`, this means it came with the JDK. If it starts with something else, it likely shows where it came from using the website name in reverse. For example, `com.amazon.javabook` tells us the code came from Amazon.com. After the website name, you can add whatever you want. For example, `com.amazon.java.my.name` also came from `Amazon.com`. Java calls more detailed packages *child packages*. The package `com.amazon.javabook` is a child package of `com.amazon`. You can tell because it's longer and thus more specific.

You'll see package names on the exam that don't follow this convention. Don't be surprised to see package names like `a.b.c`. The rule for package names is that they are mostly letters or numbers separated by periods

(`.`). Technically, you're allowed a couple of other characters between the periods (`.`). The rules are the same as for variable names, which you'll see in Chapter 2. The exam may try to trick you with invalid variable names. Luckily, it doesn't try to trick you by giving invalid package names.

In the following sections, we'll look at imports with wildcards, naming conflicts with imports, how to create a package of your own, and how the exam formats code.

Wildcards

Classes in the same package are often imported together. You can use a shortcut to `import` all the classes in a package.

```
import java.util.*;    // imports java.util.Random among other things
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();
      System.out.println(r.nextInt(10));
   }
}
```

In this example, we imported `java.util.Random` and a pile of other classes. The `*` is a wildcard that matches all classes in the package. Every class in the `java.util` package is available to this program when Java compiles it. It doesn't `import` child packages, fields, or methods; it imports only classes. (There is a special type of `import` called the *static import* that imports other types, which you'll learn more about in Chapter 7.)

You might think that including so many classes slows down your program execution, but it doesn't. The compiler figures out what's actually needed. Which approach you choose is personal preference—or team preference if you are working with others on a team. Listing the classes used makes the code easier to read, especially for new programmers. Using the wildcard can shorten the `import` list. You'll see both approaches on the exam.

Redundant Imports

Wait a minute! We've been referring to `System` without an `import`, and Java found it just fine. There's one special package in the Java world called `java.lang`. This package is special in that it is automatically imported. You can type this package in an `import` statement, but you don't have to. In the following code, how many of the imports do you think are redundant?

```
1:  import java.lang.System;
2:  import java.lang.*;
3:  import java.util.Random;
4:  import java.util.*;
5:  public class ImportExample {
6:     public static void main(String[] args) {
7:        Random r = new Random();
8:        System.out.println(r.nextInt(10));
9:     }
10: }
```

The answer is that three of the imports are redundant. Lines 1 and 2 are redundant because everything in `java.lang` is automatically considered to be imported. Line 4 is also redundant in this example because `Random` is already imported from `java.util.Random`. If line 3 wasn't present, `java.util.*` wouldn't be redundant, though, since it would cover importing `Random`.

Another case of redundancy involves importing a class that is in the same package as the class importing it. Java automatically looks in the current package for other classes.

Let's take a look at one more example to make sure you understand the edge cases for imports. For this example, `Files` and `Paths` are both in the package `java.nio.file`. You don't need to memorize this package for the 1Z0-815 exam (but you should know it for the 1Z0-816 exam). When testing your understanding of packages and imports, the 1Z0-815 exam may use packages you may never have seen before. The question will let you know which package the class is in if you need to know that in order to answer the question.

What imports do you think would work to get this code to compile?

```
public class InputImports {
   public void read(Files files) {
      Paths.get("name");
   }
}
```

There are two possible answers. The shorter one is to use a wildcard to `import` both at the same time.

```
import java.nio.file.*;
```

The other answer is to `import` both classes explicitly.

```
import java.nio.file.Files;
import java.nio.file.Paths;
```

Now let's consider some imports that don't work.

```
import java.nio.*;         // NO GOOD - a wildcard only matches
                           // class names, not "file.Files"

import java.nio.*.*;       // NO GOOD - you can only have one wildcard
                           // and it must be at the end

import java.nio.file.Paths.*; // NO GOOD - you cannot import methods
                              // only class names
```

Naming Conflicts

One of the reasons for using packages is so that class names don't have to be unique across all of Java. This means you'll sometimes want to `import` a class that can be found in multiple places. A common example of this is the `Date` class. Java provides implementations of `java.util.Date` and `java.sql.Date`. This is another example where you don't need to know the package names for the 1Z0-815 exam—they will be provided to you. What `import` could we use if we want the `java.util.Date` version?

```
public class Conflicts {
   Date date;
   // some more code
}
```

The answer should be easy by now. You can write either `import java.util.*;` or `import java.util.Date;`. The tricky cases come about when other imports are present.

```
import java.util.*;
import java.sql.*; // causes Date declaration to not compile
```

When the class is found in multiple packages, Java gives you a compiler error.

```
error: reference to Date is ambiguous
   Date date;
   ^
   both class java.sql.Date in java.sql and class java.util.Date in java.util match
```

In our example, the solution is easy—remove the `import java.sql.Date` that we don't need. But what do we do if we need a whole pile of other classes in the `java.sql` package?

```
import java.util.Date;
import java.sql.*;
```

Ah, now it works. If you explicitly `import` a class name, it takes precedence over any wildcards present. Java thinks, "The programmer really wants me to assume use of the `java.util.Date` class."

One more example. What does Java do with "ties" for precedence?

```
import java.util.Date;
import java.sql.Date;
```

Java is smart enough to detect that this code is no good. As a programmer, you've claimed to explicitly want the default to be both the `java.util.Date` and `java.sql.Date` implementations. Because there can't be two defaults, the compiler tells you the following:

```
error: reference to Date is ambiguous
   Date date;
   ^
   both class java.util.Date in java.util and class java.sql.Date in java.sql match
```

IF YOU REALLY NEED TO USE TWO CLASSES WITH THE SAME NAME

Sometimes you really do want to use `Date` from two different packages. When this happens, you can pick one to use in the `import` and use the other's fully qualified class name [the package name, a period (`.`), and the class name] to specify that it's special. Here's an example:

```
import java.util.Date;

public class Conflicts {
   Date date;
   java.sql.Date sqlDate;

}
```

Or you could have neither with an `import` and always use the fully qualified class name.

```
public class Conflicts {
   java.util.Date date;
   java.sql.Date sqlDate;

}
```

Creating a New Package

Up to now, all the code we've written in this chapter has been in the *default package*. This is a special unnamed package that you should use only for throwaway code. You can tell the code is in the default package, because there's no package name. On the exam, you'll see the default package used a lot to save space in code listings. In real life, always name your packages to avoid naming conflicts and to allow others to reuse your code.

Now it's time to create a new package. The directory structure on your computer is related to the package name. In this section, just read along. We will cover how to compile and run the code in the next section.

Suppose we have these two classes in the `C:\temp` directory:

```
package packagea;
public class ClassA {
}

package packageb;
import packagea.ClassA;
public class ClassB {
   public static void main(String[] args) {
      ClassA a;
      System.out.println("Got it");
   }
}
```

When you run a Java program, Java knows where to look for those package names. In this case, running from `C:\temp` works because both `packagea` and `packageb` are underneath it.

What do you think happens if you run `java packageb/ClassB.java`? This does not work. Remember that you can use the `java` command to run a file directly only when that program is contained within a single file. Here, `ClassB.java` relies on `ClassA`.

Compiling and Running Code with Packages

You'll learn Java much more easily by using the command line to compile and test your examples. Once you know the Java syntax well, you can switch to an IDE. But for the exam, your goal is to know details about the language and not have the IDE hide them for you.

Follow this example to make sure you know how to use the command line. If you have any problems following this procedure, post a question in the Beginning Java forum at CodeRanch (www.coderanch.com/forums/f-33/java). Describe what you tried and what the error said.

The first step is to create the two files from the previous section. Table 1.2 shows the expected fully qualified filenames and the command to get into the directory for the next steps.

TABLE 1.2 Setup procedure by operating system

Step	Windows	Mac/Linux
1. Create first class.	`C:\temp\packagea\ClassA.java`	`/tmp/packagea/ClassA.java`
2. Create second class.	`C:\temp\packageb\ClassB.java`	`/tmp/packageb/ClassB.java`
3. Go to directory.	`cd C:\temp`	`cd /tmp`

Now it is time to compile the code. Luckily, this is the same regardless of the operating system. To compile, type the following command:

```
javac packagea/ClassA.java packageb/ClassB.java
```

If this command doesn't work, you'll get an error message. Check your files carefully for typos against the provided files. If the command does work, two new files will be created: `packagea/ClassA.class` and `packageb/ClassB.class`.

COMPILING WITH WILDCARDS

You can use an asterisk to specify that you'd like to include all Java files in a directory. This is convenient when you have a lot of files in a package. We can rewrite the previous `javac` command like this:

```
javac packagea/*.java packageb/*.java
```

However, you cannot use a wildcard to include subdirectories. If you were to write `javac *.java`, the code in the packages would not be picked up.

Now that your code has compiled, you can run it by typing the following command:

```
java packageb.ClassB
```

If it works, you'll see `Got it` printed. You might have noticed that we typed `ClassB` rather than `ClassB.class`. As discussed earlier, you don't pass the extension when running a program.

Figure 1.1 shows where the `.class` files were created in the directory structure.

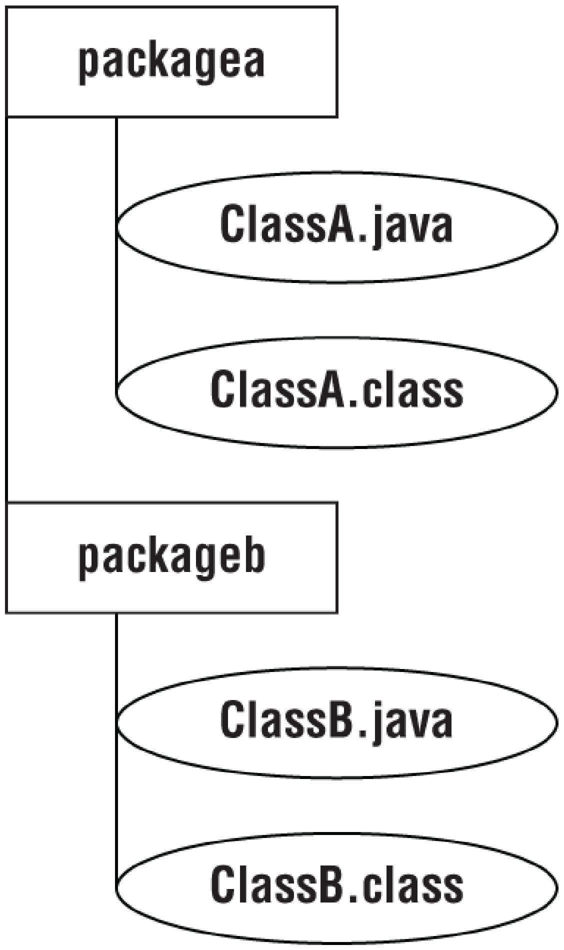

FIGURE 1.1 Compiling with packages

Using an Alternate Directory

By default, the `javac` command places the compiled classes in the same directory as the source code. It also provides an option to place the class files into a different directory. The `-d` option specifies this target directory.

NOTE

Java options are case sensitive. This means you cannot pass `-D` instead of `-d` .

If you are following along, delete the `ClassA.class` and `ClassB.class` files that were created in the previous section.

Where do you think this command will create the file `ClassA.class` ?

```
javac -d classes packagea/ClassA.java packageb/ClassB.java
```

The correct answer is `classes/packagea/ClassA.class` . The package structure is preserved under the requested target directory. Figure 1.2 shows this new structure.

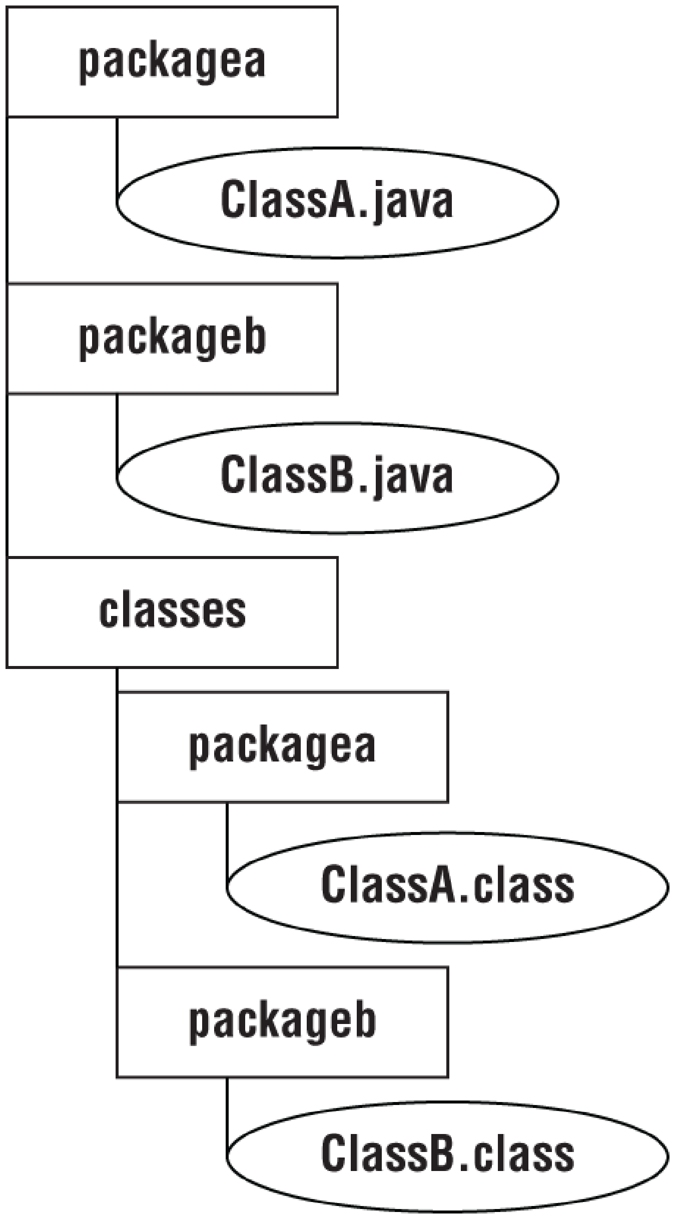

FIGURE 1.2 **Compiling with packages and directories**

To run the program, you specify the classpath so Java knows where to find the classes. There are three options you can use. All three of these do the same thing:

```
java -cp classes packageb.ClassB
java -classpath classes packageb.ClassB
java --class-path classes packageb.ClassB
```

Notice that the last one requires two dashes (`--`), while the first two require one dash (`-`). If you have the wrong number of dashes, the program will not run.

THREE CLASSPATH OPTIONS

You might wonder why there are three options for the classpath. The `-cp` option is the short form. Developers frequently choose the short form because we are lazy typists. The `-classpath` and `--class-path` versions can be clearer to read but require more typing. The exam can use any of these, so be sure to learn all three.

Table 1.3 and Table 1.4 review the options you need to know for the exam. In Chapter 11, "Modules," you will learn additional options specific to modules.

TABLE 1.3 Options you need to know for the exam: `javac`

Option	Description
`-cp <classpath>` `-classpath <classpath>` `--class-path <classpath>`	Location of classes needed to compile the program
`-d <dir>`	Directory to place generated class files

TABLE 1.4 Options you need to know for the exam: `java`

Option	Description
`-cp <classpath>` `-classpath <classpath>` `--class-path <classpath>`	Location of classes needed to run the program

Compiling with JAR Files

Just like the `classes` directory in the previous example, you can also specify the location of the other files explicitly using a classpath. This technique is useful when the class files are located elsewhere or in special JAR files. A *Java archive* (JAR) file is like a zip file of mainly Java class files.

On Windows, you type the following:

```
java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass
```

And on macOS/Linux, you type this:

```
java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass
```

The period (`.`) indicates you want to include the current directory in the classpath. The rest of the command says to look for loose class files (or packages) in `someOtherLocation` and within `myJar.jar` . Windows uses semicolons (`;`) to separate parts of the classpath; other operating systems use colons.

Just like when you're compiling, you can use a wildcard (`*`) to match all the JARs in a directory. Here's an example:

```
java -cp "C:\temp\directoryWithJars\*" myPackage.MyClass
```

This command will add all the JARs to the classpath that are in `directoryWithJars` . It won't include any JARs in the classpath that are in a subdirectory of `directoryWithJars` .

Creating a JAR File

Some JARs are created by others, such as those downloaded from the Internet or created by a teammate. Alternatively, you can create a JAR file yourself. To do so, you use the `jar` command. The simplest commands create a `jar` containing the files in the current directory. You can use the short or long form for each option.

```
jar -cvf myNewFile.jar .
jar --create --verbose --file myNewFile.jar .
```

Alternatively, you can specify a directory instead of using the current directory.

```
jar -cvf myNewFile.jar -C dir .
```

There is no long form of the `-C` option. Table 1.5 lists the options you need to use the `jar` command to create a `jar` file. In Chapter 11, you will learn another option specific to modules.

TABLE 1.5 **Options you need to know for the exam: `jar`**

Option	Description
`-c` `--create`	Creates a new JAR file
`-v` `--verbose`	Prints details when working with JAR files

`-f <fileName>`
`--file <fileName>`
	JAR filename

`-C <directory>`	Directory containing files to be used to create the JAR

Running a Program in One Line with Packages

You can use single-file source-code programs from within a package as long as they rely only on classes supplied by the JDK. This code meets the criteria.

```
package singleFile;

import java.util.*;

public class Learning {
   private ArrayList list;
   public static void main(String[] args) {
      System.out.println("This works!");
   }
}
```

You can run either of these commands:

```
java Learning.java            // from within the singleFile directory
java singleFile/Learning.java // from the directory above singleFile
```

Ordering Elements in a Class

Now that you've seen the most common parts of a class, let's take a look at the correct order to type them into a file. Comments can go anywhere in the code. Beyond that, you need to memorize the rules in Table 1.6.

TABLE 1.6 Order for declaring a `class`

Element	Example	Required?	Where does it go?
Package declaration	`package abc;`	No	First line in the file
Import statements	`import java.util.*;`	No	Immediately after the package (if present)
Class declaration	`public class C`	Yes	Immediately after the import (if any)
Field declarations	`int value;`	No	Any top-level element in a class
Method declarations	`void method()`	No	Any top-level element in a class

Let's look at a few examples to help you remember this. The first example contains one of each element:

```
package structure;      // package must be first non-comment
import java.util.*;     // import must come after package
public class Meerkat {  // then comes the class
   double weight;       // fields and methods can go in either order
   public double getWeight() {
      return weight; }
   double height;    // another field - they don't need to be together
}
```

So far, so good. This is a common pattern that you should be familiar with. How about this one?

```
/* header */
package structure;
// class Meerkat
public class Meerkat { }
```

Still good. We can put comments anywhere, and imports are optional. In the next example, we have a problem:

```
import java.util.*;
package structure;       // DOES NOT COMPILE
String name;             // DOES NOT COMPILE
public class Meerkat { } // DOES NOT COMPILE
```

There are two problems here. One is that the `package` and `import` statements are reversed. Though both are optional, `package` must come before `import` if present. The other issue is that a field attempts a declaration outside a class. This is not allowed. Fields and methods must be within a class.

Got all that? Think of the acronym PIC (picture): package, import, and class. Fields and methods are easier to remember because they merely have to be inside a class.

You need to know one more thing about class structure for the 1Z0-815 exam: multiple classes can be defined in the same file, but only one of them is allowed to be public. The public class matches the name of the file. For example, these two classes must be in a file named `Meerkat.java`:

```
public class Meerkat { }
class Paw { }
```

A file is also allowed to have neither class be public. As long as there isn't more than one public class in a file, it is okay.

Now you know how to create and arrange a `class`. Later chapters will show you how to create classes with more powerful operations.

Code Formatting on the Exam

Not all questions will include package declarations and imports. Don't worry about missing package statements or imports unless you are asked about them. The following are common cases where you don't need to check the imports:

- Code that begins with a class name
- Code that begins with a method declaration
- Code that begins with a code snippet that would normally be inside a class or method
- Code that has line numbers that don't begin with 1

This point is so important that we are going to reinforce it with an example. Does this code compile?

```
public class MissingImports {
   Date date;
   public void today() {}
}
```

Yes! The question was not about imports, so you have to assume that `import java.util` is present.

On the other hand, a question that asks you about packages, imports, or the correct order of elements in a class is giving you clues that the question is virtually guaranteed to be testing you on these topics! Also note that imports will be not removed to save space if the package statement is present. This is because imports go after the package statement.

You'll see code that doesn't have a `main()` method. When this happens, assume any necessary plumbing code like the `main()` method and class definition were written correctly. You're just being asked if the part of the code you're shown compiles when dropped into valid surrounding code.

Another thing the exam does to save space is to merge code on the same line. You should expect to see code like the following and to be asked whether it compiles. (You'll learn about `ArrayList` in Chapter 5—assume that part is good for now.)

```
6: public void getLetter(ArrayList list) {
7:    if (list.isEmpty()) { System.out.println("e");
8:    } else { System.out.println("n");
9: }   }
```

The answer here is that it does compile because the line break between the `if` statement and `println()` is not necessary. Additionally, you still get to assume the necessary class definition and imports are present. Now, what about this one? Does it compile?

```
1: public class LineNumbers {
2:    public void getLetter(ArrayList list) {
3:       if (list.isEmpty()) { System.out.println("e");
4:       } else { System.out.println("n");
5: }   } }
```

For this one, you would answer "Does not compile." Since the code begins with line 1, you don't get to assume that valid imports were provided earlier. The exam will let you know what package classes are in unless they're covered in the objectives. You'll be expected to know that `ArrayList` is in `java.util`—at least you will once you get to Chapter 5 of this book!

NOTE

Remember that extra whitespace doesn't matter in Java syntax. The exam may use varying amounts of whitespace to trick you.

Summary

The Java Development Kit (JDK) is used to do software development. It contains the compiler (`javac`), which turns source code into bytecode. It also contains the Java Virtual Machine (JVM) launcher (`java`), which launches the JVM and then calls the code. Application programming interfaces (APIs) are available to call reusable pieces of code.

Java code is object-oriented, meaning all code is defined in classes. Access modifiers allow classes to encapsulate data. Java is platform independent, compiling to bytecode. It is robust and simple by not providing pointers or operator overloading. Java is secure because it runs inside a virtual machine. Finally, the language facilitates multithreaded programming and strives for backward compatibility.

Java classes consist of members called fields and methods. An object is an instance of a Java class. There are three styles of comments: a single-line comment (`//`), a multiline comment (`/* */`), and a Javadoc comment (`/** */`).

Java begins program execution with a `main()` method. The most common signature for this method run from the command line is `public static void main(String[] args)` . Arguments are passed in after the class name, as in `java NameOfClass firstArgument` . Arguments are indexed starting with 0.

Java code is organized into folders called packages. To reference classes in other packages, you use an `import` statement. A wildcard ending an `import` statement means you want to `import` all classes in that package. It does not include packages that are inside that one. The package `java.lang` is special in that it does not need to be imported.

For some class elements, order matters within the file. The package statement comes first if present. Then come the `import` statements if present. Then comes the class declaration. Fields and methods are allowed to be in any order within the class.

Exam Essentials

Identify benefits of Java. Benefits of Java include object-oriented design, encapsulation, platform independence, robustness, simplicity, security, multithreading, and backward compatibility.

Define common acronyms. The JDK stands for Java Development Kit and contains the compiler and JVM launcher. The JVM stands for Java Virtual Machine, and it runs bytecode. API is an application programming interface, which is code that you can call.

Be able to write code using a *main()* method. A `main()` method is usually written as `public static void main(String[] args`). Arguments are referenced starting with `args[0]`. Accessing an argument that wasn't passed in will cause the code to throw an exception.

Understand the effect of using packages and imports. Packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not look at subdirectories. In the event of a conflict, class name imports take precedence.

Be able to recognize misplaced statements in a class. Package and `import` statements are optional. If present, both go before the class declaration in that order. Fields and methods are also optional and are allowed in any order within the class declaration.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which of the following are true statements? (Choose all that apply.)
    1. Java allows operator overloading.
    2. Java code compiled on Windows can run on Linux.
    3. Java has pointers to specific locations in memory.
    4. Java is a procedural language.
    5. Java is an object-oriented language.
    6. Java is a functional programming language.
2. Which of the following are true? (Choose all that apply.)
    1. `javac` compiles a `.class` file into a `.java` file.
    2. `javac` compiles a `.java` file into a `.bytecode` file.
    3. `javac` compiles a `.java` file into a `.class` file.
    4. `java` accepts the name of the class as a parameter.
    5. `java` accepts the filename of the `.bytecode` file as a parameter.
    6. `java` accepts the filename of the `.class` file as a parameter.
3. Which of the following are true if this command completes successfully assuming the `CLASSPATH` is not set? (Choose all that apply.)
    1. java MyProgram.java
    1. A `.class` file is created.
    2. `MyProgram` can reference classes in the package `com.sybex.book`.
    3. `MyProgram` can reference classes in the package `java.lang`.
    4. `MyProgram` can reference classes in the package `java.util`.
    5. None of the above. The program needs to be run as `java MyProgram`.
4. Given the following classes, which of the following can independently replace `INSERT IMPORTS HERE` to make the code compile? (Choose all that apply.)

    1.
       ```
       package aquarium;
       public class Tank { }
       ```

2. `package aquarium.jellies;`
   ```
   public class Jelly { }
   ```
3. `package visitor;`
   ```
   INSERT IMPORTS HERE
   public class AquariumVisitor {
     public void admire(Jelly jelly) { } }
   ```
1. `import aquarium.*;`
2. `import aquarium.*.Jelly;`
3. `import aquarium.jellies.Jelly;`
4. `import aquarium.jellies.*;`
5. `import aquarium.jellies.Jelly.*;`
6. None of these can make the code compile.

5. Which are included in the JDK? (Choose all that apply.)
   1. `javac`
   2. Eclipse
   3. JVM
   4. `javadoc`
   5. `jar`
   6. None of the above
6. Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?
   1. `package aquarium;`
      ```
      public class Water { }
      ```
   2. `package aquarium;`
      ```
      import java.lang.*;
      import java.lang.System;
      import aquarium.Water;
      import aquarium.*;
      public class Tank {
        public void print(Water water) {
           System.out.println(water); } }
      ```
   1. 0
   2. 1
   3. 2
   4. 3
   5. 4
   6. Does not compile
7. Given the following classes, which of the following snippets can independently be inserted in place of `INSERT IMPORTS HERE` and have the code compile? (Choose all that apply.)
   1. `package aquarium;`
      ```
      public class Water {
         boolean salty = false;
      }
      ```
   2.
      ```
      package aquarium.jellies;
      public class Water {
      ```

```
      boolean salty = true;
   }

   package employee;
   INSERT IMPORTS HERE
   public class WaterFiller {
      Water water;
   }
```

1. `import aquarium.*;`
2. `import aquarium.Water;`
   `import aquarium.jellies.*;`
3. `import aquarium.*;`
   `import aquarium.jellies.Water;`
4. `import aquarium.*;`
   `import aquarium.jellies.*;`
5. `import aquarium.Water;`
   `import aquarium.jellies.Water;`
6. None of these imports can make the code compile.

8. Given the following command, which of the following classes would be included for compilation? (Choose all that apply.)
   1. javac *.java
   1. `Hyena.java`
   2. `Warthog.java`
   3. `land/Hyena.java`
   4. `land/Warthog.java`
   5. `Hyena.groovy`
   6. `Warthog.groovy`
9. Given the following class, which of the following calls print out `Blue Jay` ? (Choose all that apply.)
   1. `public class BirdDisplay {`
      ```
         public static void main(String[] name) {
            System.out.println(name[1]);
      ```
   2. `} }`
   1. `java BirdDisplay Sparrow Blue Jay`
   2. `java BirdDisplay Sparrow "Blue Jay"`
   3. `java BirdDisplay Blue Jay Sparrow`
   4. `java BirdDisplay "Blue Jay" Sparrow`
   5. `java BirdDisplay.class Sparrow "Blue Jay"`
   6. `java BirdDisplay.class "Blue Jay" Sparrow`
10. Which of the following are legal entry point methods that can be run from the command line? (Choose all that apply.)
    1. `private static void main(String[] args)`
    2. `public static final main(String[] args)`
    3. `public void main(String[] args)`
    4. `public static void test(String[] args)`
    5. `public static void main(String[] args)`
    6. `public static main(String[] args)`

11. Which of the following are true statements about Java? (Choose all that apply.)
    1. Bug-free code is guaranteed.
    2. Deprecated features are never removed.
    3. Multithreaded code is allowed.
    4. Security is a design goal.
    5. Sideways compatibility is a design goal.
12. Which options are valid on the `javac` command without considering module options? (Choose all that apply.)
    1. `-c`
    2. `-C`
    3. `-cp`
    4. `-CP`
    5. `-d`
    6. `-f`
    7. `-p`
13. Which options are valid on the `java` command without considering module options? (Choose all that apply.)
    1. `-c`
    2. `-C`
    3. `-cp`
    4. `-d`
    5. `-f`
    6. `-p`
14. Which options are valid on the `jar` command without considering module options? (Choose all that apply.)
    1. `-c`
    2. `-C`
    3. `-cp`
    4. `-d`
    5. `-f`
    6. `-p`
15. What does the following code output when run as `java Duck Duck Goose`?
    1.
       ```
       public class Duck {
           public void main(String[] args) {
              for (int i = 1; i <= args.length; i++)
                 System.out.println(args[i]);
       } }
       ```
    1. `Duck Goose`
    2. `Duck ArrayIndexOutOfBoundsException`
    3. `Goose`
    4. `Goose ArrayIndexOutOfBoundsException`
    5. None of the above
16. Suppose we have the following class in the file `/my/directory/named/A/Bird.java`. Which of the answer options re-

places `INSERT CODE HERE` when added independently if we compile from `/my/directory`? (Choose all that apply.)

1.
   ```
   INSERT CODE HERE
   public class Bird { }
   ```

1. `package my.directory.named.a;`
2. `package my.directory.named.A;`
3. `package named.a;`
4. `package named.A;`
5. `package a;`
6. `package A;`

17. Which of the following are true? (Choose all that apply.)

1.
   ```
   public class Bunny {
       public static void main(String[] x) {
           Bunny bun = new Bunny();
   } }
   ```

1. `Bunny` is a class.
2. `bun` is a class.
3. `main` is a class.
4. `Bunny` is a reference to an object.
5. `bun` is a reference to an object.
6. `main` is a reference to an object.
7. The `main()` method doesn't run because the parameter name is incorrect.

18. Which answer options represent the order in which the following statements can be assembled into a program that will compile successfully? (Choose all that apply.)

1.
   ```
   X: class Rabbit {}
   Y: import java.util.*;
   Z: package animals;
   ```

1. X, Y, Z
2. Y, Z, X
3. Z, Y, X
4. Y, X
5. Z, X
6. X, Z

19. Which are not available for download from Oracle for Java 11? (Choose all that apply.)

1. JDK
2. JRE
3. Eclipse
4. All of these are available from Oracle.

20. Which are valid ways to specify the classpath when compiling? (Choose all that apply.)

1. `-cp`
2. `-classpath`
3. `--classpath`
4. `-class-path`

5. `--class-path`