

Chapter 1

Building Blocks

OCF EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Handling Date, Time, Text, Numeric and Boolean Values**
 - Use primitives and wrapper classes. Evaluate arithmetic and boolean expressions, using the Math API and by applying precedence rules, type conversions, and casting.
 - **Using Object-Oriented Concepts in Java**
 - Declare and instantiate Java objects including nested class objects, and explain the object life-cycle including creation, reassigning references, and garbage collection.
 - Understand variable scopes, apply encapsulation, and create immutable objects. Use local variable type inference.
-

Welcome to the beginning of your journey to achieve a Java 21 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 need for the exam. If you've never written a Java program before, we recommend you pick up an introductory book on Java 8 or higher. Examples include *Head First Java, 3rd Edition* (O'Reilly Media, 2022) and *Beginning Programming with Java for Dummies* (For Dummies, 2021). Then come back to this certification study guide.

As the old saying goes, you have to learn how to walk before you can run. Likewise, you have to learn the basics of Java before you can build complex programs. In this chapter, we present the basics of Java packages, classes, variables, and data types, along with the aspects of each that you need to know for the exam. For example, you might use Java every day but be unaware that you cannot create a variable called `3dMap` or `this`. The exam expects you to know and understand the rules behind these principles. While most of this chapter should be review, there may be aspects of the Java language that are new to you since they don't come up in practical use often.

Learning About the Environment

The Java environment consists of a number of technologies. In the following sections, we go over the key terms and acronyms you need to know 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 commands include the following:

- `javac`: Converts `.java` source files into `.class` bytecode
- `java`: Executes the program
- `jar`: Packages files together
- `javadoc`: Generates documentation

The `javac` program generates instructions in a special format called *bytecode* that the `java` command can run. 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 within your particular operating system and hardware.

Where Did the JRE Go?

In Java 8 and earlier, 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. Now, 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.

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 Visual Studio Code.

Downloading a JDK

Every six months, Oracle releases a new version of Java. Java 21 came out in September 2023. This means Java 21 will not be the latest version when you download the JDK to study for the exam. However, you should still use Java 21 to study with since this is a Java 21 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!

You can download Oracle's JDK on the Oracle website, using the same account you use to register for the exam. There are many JDKs available, the most popular of which, besides Oracle's JDK, is OpenJDK.

Many versions of Java include *preview features* that are off by default but that you can enable. Preview features are not on the exam. To avoid confusion about when a feature was added to the language, we will say "was officially introduced in" to denote when it was moved out of preview.

Check 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 version number 21.

Understanding the 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. In later chapters, you see other building blocks such as interfaces, records, and enums.

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 look at fields, methods, and comments. We 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's job to

create and arrange these elements in such a way that the resulting code is useful and, ideally, easy for other programmers to understand.

The simplest Java class you can write looks like this:

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

Java calls a word with special meaning a *keyword*, which we've marked bold in the previous snippet. Throughout the book, we often bold parts of code snippets to call attention to them. Line 1 includes the **public** keyword, which allows other classes to use it. 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 declare the type of that variable to be **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 we 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, we define a 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 that 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.

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) {  
    return 10;  
}
```

The method name is `numberVisitors`. There's one parameter named `month`, which is of type `int`, which is a numeric type. Therefore, the method signature is `numberVisitors(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. While the exam creators are trying to make the code harder to read, they still use comments to call attention to line numbers. We hope you use comments in your own code. There are three types of comments in Java. The first is 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, this is 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 each 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 looking 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, even though it takes up only one line.

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 and Source Files

Most of the time, each Java class is defined in its own `.java` file. In this chapter, the only top-level type is a class. A *top-level type* is a data structure that can be defined independently within a source file. For the majority of the book, we work with classes as the top-level type, but in [Chapter 7](#), “Beyond Classes,” we present other top-level types, as well as nested types.

A top-level class is often `public`, which means any code can call it. Interestingly, Java does not require that the type be `public`. For example, this class is just fine:

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

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

```
1: public class Animal {  
2:     private String name;  
3: }  
4: class Animal2 {}
```

If you do have a `public` type, it needs to match the filename. The declaration `public class Animal2` would not compile in a file named `Animal.java`. In [Chapter 5](#), “Methods,” we discuss what access options are available other than `public`.

Noticing a pattern yet? This chapter includes numerous references to topics that we go into in more detail in later chapters. If you’re an experienced Java developer, you’ll notice we keep the examples and rules simple in this chapter. Don’t worry; we have the rest of the book to present more rules and complicated edge cases!

Writing a *main()* Method

A Java program begins execution with its `main()` method. In this section, you learn how to create one, pass a parameter, and run a program. The `main()` method is often called an entry point into the program, because it is the starting point that the JVM looks for when it begins running a new program.

Creating a *main()* Method

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

```
1: public class Zoo {  
2:     public static void main(String[] args) {  
3:         System.out.println("Hello World");  
4:     }  
5: }
```

This code prints `Hello world`. To compile and execute this code, type it into a file called `Zoo.java` and execute the following:

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

If it prints `Hello world`, you were successful. If you do get error messages, check that you've installed the Java 21 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:

<https://www.coderanch.com/forums/f-33/java>

To compile Java code with the `javac` command, the file must have the extension `.java`. The name of the file must match the name of the `public` class. The result is a file of bytecode with 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.class`.

The rules for what a Java 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 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.
- If the Java class is an entry point for the program, it must contain a valid `main()` method.

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 full access from anywhere in the program. You learn more about access modifiers in [Chapter 5](#).

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 doesn't have the right keywords, you'll get an error trying to run it. You see `static` again in [Chapter 6](#), "Class Design."

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 explore return types in [Chapter 5](#) as well. (Are you excited for [Chapter 5](#) yet?)

Finally, we arrive at the `main()` method's parameter list, represented as an array of `java.lang.String` objects. You can use any valid variable name along with any of these three formats:

```
String[] args
String options[]
String... friends
```

The compiler accepts any of these. The variable name `args` is common because it 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 learn about `String` in this chapter. Arrays are in [Chapter 4](#), “Core APIs,” and *varargs* are in [Chapter 5](#).

Optional Modifiers in *main()* Methods

While most modifiers, such as `public` and `static`, are required for `main()` methods, there are some optional modifiers allowed.

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

In this example, both `final` modifiers are optional, and the `main()` method is a valid entry point with or without them. We cover the meaning of `final` methods and parameters in [Chapter 6](#).

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
```

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 learn about exceptions in [Chapter 11](#), “Exceptions and Localization.”)

```
Zoo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
    Index 1 out of bounds for length 1
    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.

Single-File Source Code

If you get tired of typing both `javac` and `java` every time you want to try a code example, there's a shortcut. You can instead run this:

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

There is a key difference here. When compiling first, you omitted the file extension when running java. When skipping the explicit compilation step, you include this extension. This feature is called launching *single-file source-code* programs and is useful for testing or for small programs. The name cleverly tells you that it is designed for when your program is one file.

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 NumberPicker {
    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:

```
error: cannot find symbol
```

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. A *statement* is an instruction, and 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 the code to compile.

```
import java.util.Random; // import tells us where to find Random
public class NumberPicker {
    public static void main(String[] args) {
        Random r = new Random();
        System.out.println(r.nextInt(10)); // 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.

In [Chapter 5](#), we cover another type of import referred to as a static import. It allows you to make static members of a class known, often so you can use variables and method names without having to keep specifying the class name.

Packages

As you saw 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 Street, Apartment 9. The mail carrier first brings the letter to 123 Main Street. Then the carrier looks for the mailbox for apartment 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. For example, if it begins with `java`, 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.wiley.javabook` tells us the code is associated with the wiley.com website or organization. After the website name, you can add whatever you want. For example, `com.wiley.java.my.name` also came from wiley.com. Java calls more detailed packages *child packages*. The package `com.wiley.javabook` is a child package of `com.wiley`. 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 (`.`). You can even use package names of websites you don't own if you want to, such as `com.wiley`, although people reading your code might be confused! The rules are the same as for variable names, which you see later in this chapter. 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 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 NumberPicker {
    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. The `import` statement doesn't bring in child packages, fields, or methods; it imports only classes directly under the package. Let's say you wanted to use the class `AtomicInteger` (you learn about that one in [Chapter 13](#), "Concurrency") in the `java.util.concurrent.atomic` package. Which import or imports support this?

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
```

Only the last import allows the class to be recognized because child packages are not included with the first two.

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` every time we printed text, 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 NumberPicker {
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 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`. The 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.

Which import statements 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`. What import statement can 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 name is found in multiple packages, Java gives you a compiler error. In our example, the solution is easy—remove the `import java.sql.*` 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 imports are ambiguous.

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` statement and use the other's *fully qualified class name*. Or you can drop both `import` statements 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 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.

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. Describe what you tried and what the error said.

<https://www.coderanch.com/forums/f-33/java>

The first step is to create the two files from the previous section. [Table 1.1](#) shows the expected fully qualified filenames and the command to get into the directory for the next steps.

TABLE 1.1 Setup procedure by operating system

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

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.

An image illustrates the directory structure of packages. It includes two packages `a` and `b`. Package `a` includes `class A dot java` and `class A dot class`. Package `b` includes `class B dot java` and `class B dot class`.

[FIGURE 1.1](#) Compiling with packages

Compiling to Another 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.

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 in `classes/packagea/ClassA.class`. The package structure is preserved under the requested target directory. [Figure 1.2](#) shows this new structure.

An image illustrates the target directory structure of the package. Package a includes class A dot java, package b includes class B dot java, classes includes package a of class A dot class, and package b of class B dot class.

[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.

[Table 1.2](#) and [Table 1.3](#) review the options you need to know for the exam. There are *many* other options available! And in [Chapter 12](#), “Modules,” you learn additional options specific to modules.

[TABLE 1.2](#) Important javac options

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

TABLE 1.3 Important java options

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

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 1.4](#).

TABLE 1.4 Order for declaring a class

Element	Example	Required?	Where does it go?
Package declaration	package abc;	No	First line in the file (excluding comments or blank lines)
import statements	import java.util.*;	No	Immediately after the package (if present)
Top-level type declaration	public class C	Yes	Immediately after the imports (if any)
Field declarations	int value;	No	Any top-level element within a class
Method declarations	void method()	No	Any top-level element within 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, blank lines are ignored, 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. Although 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.

Throughout this book, if you see two public classes in a code snippet or question, you can assume they are in different files unless it specifically says they are in the same .java file.

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

Creating Objects

Our programs wouldn't be able to do anything useful if we didn't have the ability to create new objects. Remember that an object is an instance of a class. In the following sections, we look at constructors, object fields, instance initializers, and the order in which values are initialized.

Calling Constructors

To create an instance of a class, all you have to do is write `new` before the class name and add parentheses after it. Here's an example:

```
Park p = new Park();
```

First you declare the type that you'll be creating (`Park`) and give the variable a name (`p`). This gives Java a place to store a reference to the object. Then you write `new Park()` to actually create the object.

`Park()` looks like a method since it is followed by parentheses. It's called a *constructor*, which is a special type of method that creates a new object. Now it's time to define a constructor of your own.

```
public class Chick {
    public Chick() {
        System.out.println("in constructor");
    }
}
```

There are two key points to note about the constructor: the name of the constructor matches the name of the class, and there's no return type. You may see a method like this on the exam:

```
public class Chick {
    public void Chick() { } // NOT A CONSTRUCTOR
}
```

When you see a method name beginning with a capital letter and having a return type, pay special attention to it. It is *not* a constructor since there's a return type. It's a regular method that does compile but will not be called when you write `new Chick()`.

The purpose of a constructor is to initialize fields, although you can put any code in there. Another way to initialize fields is to do so directly on the line on which they're declared. This example shows both approaches:

```
public class Chicken {
    int numEggs = 12; // initialize on line
    String name;
    public Chicken() {
```

```

        name = "Duke"; // initialize in constructor
    }
}

```

For most classes, you don't have to code a constructor—the compiler will supply a “do nothing” default constructor for you. There are some scenarios that do require you to declare a constructor. You learn all about them in [Chapter 6](#).

Reading and Writing Member Fields

It's possible to read and write instance variables directly from the caller. In this example, a mother swan lays eggs:

```

public class Swan {
    int numberEggs; // instance variable
    public static void main(String[] args) {
        Swan mother = new Swan();
        mother.numberEggs = 1; // set variable
        System.out.println(mother.numberEggs); // read variable
    }
}

```

The “caller” in this case is the `main()` method, which could be in the same class or in another class. This class sets `numberEggs` to 1 and then reads `numberEggs` directly to print it out.

You can even read values of already initialized fields on a line initializing a new field.

```

1: public class Name {
2:     String first = "Theodore";
3:     String last = "Moose";
4:     String full = first + last;
5: }

```

Lines 2 and 3 both write to fields. Line 4 both reads and writes data. It reads the fields `first` and `last`. It then writes the field `full`.

Executing Instance Initializer Blocks

When you learned about methods, you saw braces (`{}`). The code between the braces (sometimes called “inside the braces”) is called a *code block*. Anywhere you see braces is a *code block*.

Sometimes code blocks are inside a method. These are run when the method is called. Other times, code blocks appear outside a method. These are called *instance initializers*. In [Chapter 6](#), you learn how to use a static initializer.

How many blocks do you see in the following example? How many instance initializers do you see?

```
1: public class Bird {
2:     public static void main(String[] args) {
3:         { System.out.println("Feathers"); }
4:     }
5:     { System.out.println("Snowy"); }
6: }
```

There are four code blocks in this example: a class definition, a method declaration, an inner block, and an instance initializer. Counting code blocks is easy: you just count the number of pairs of braces. If there aren't the same number of open ({) and close (}) braces or they aren't defined in the proper order, the code doesn't compile. For example, you cannot use a closed brace (}) if there's no corresponding open brace ({) that it matches written earlier in the code. In programming, this is referred to as the *balanced parentheses problem*, and it often comes up in job interview questions.

When you're counting instance initializers, keep in mind that they cannot exist inside a method. Line 5 is an instance initializer, with its braces outside a method. On the other hand, line 3 is not an instance initializer, as it is called only when the `main()` method is executed. There is one additional set of braces on lines 1 and 6 that constitute the class declaration.

Following the Order of Initialization

When writing code that initializes fields in multiple places, you have to keep track of the order of initialization. This is simply the order in which different methods, constructors, or blocks are called when an instance of the class is created. We add some more rules to the order of initialization in [Chapter 6](#). In the meantime, you need to remember the following:

- Fields and instance initializer blocks are run in the order in which they appear in the file.
- The constructor runs after all fields and instance initializer blocks have run.

Let's look at an example:

```
1: public class Chick {
2:     private String name = "Fluffy";
3:     { System.out.println("setting field"); }
4:     public Chick() {
5:         name = "Tiny";
6:         System.out.println("setting constructor");
7:     }
8:     public static void main(String[] args) {
```



```
9:      Chick chick = new Chick();
10:      System.out.println(chick.name); } }
```

Running this example prints this:

```
setting field
setting constructor
Tiny
```

Let's look at what's happening here. We start with the `main()` method because that's where Java starts execution. On line 9, we call the constructor of `Chick`. Java creates a new object. First it initializes `name` to "Fluffy" on line 2. Next it executes the `println()` statement in the instance initializer on line 3. Once all the fields and instance initializers have run, Java returns to the constructor. Line 5 changes the value of `name` to "Tiny", and line 6 prints another statement. At this point, the constructor is done, and then the execution goes back to the `println()` statement on line 10.

Order matters for the fields and blocks of code. You can't refer to a variable before it has been defined.

```
{ System.out.println(name); } // DOES NOT COMPILE
private String name = "Fluffy";
```

You should expect to see a question about initialization on the exam. Let's try one more. What do you think this code prints out?

```
public class Egg {
    public Egg() {
        number = 5;
    }
    public static void main(String[] args) {
        Egg egg = new Egg();
        System.out.println(egg.number);
    }
    private int number = 3;
    { number = 4; } }
```

If you answered 5, you got it right. Fields and blocks are run first in order, setting `number` to 3 and then 4. Then the constructor runs, setting `number` to 5. You see *a lot more rules and examples* covering order of initialization in [Chapter 6](#). We only cover the basics here so you can follow the order of initialization for simple programs.

Understanding Data Types

Java applications contain two types of data: primitive types and reference types. In this section, we discuss the differences between a primitive type and a reference type.

Using Primitive Types

Java has eight built-in data types, referred to as the Java *primitive types*. These eight data types represent the building blocks for Java objects, because all Java objects are just a complex collection of these primitive data types. That said, a primitive is not an object in Java, nor does it represent an object. A primitive is just a single value in memory, such as a number or character.

The Primitive Types

The exam assumes you are well versed in the eight primitive data types, their relative sizes, and what can be stored in them. [Table 1.5](#) shows the Java primitive types together with their size in bits and the range of values that each holds.

TABLE 1.5 Primitive types

Keyword	Type	Min value	Max value	Default value	Example
boolean	true or false	n/a	n/a	false	true
byte	8-bit integral value	-128	127	0	123
short	16-bit integral value	-32,768	32,767	0	123
int	32-bit integral value	-2,147,483,648	2,147,483,647	0	123
long	64-bit integral value	-2^{63}	$2^{63} - 1$	0L	123L
float	32-bit floating-point value	n/a	n/a	0.0f	123.45f
double	64-bit floating-point value	n/a	n/a	0.0	123.456
char	16-bit Unicode value	0	65,535	\u0000	'a'

Is *String* a Primitive?

No, it is not. That said, `String` is often mistaken for a ninth primitive because Java includes built-in support for `String` literals and operators. You learn more about `String` in [Chapter 4](#), but for now, just remember it's an object, not a primitive.

There's a lot of information in [Table 1.5](#). Let's look at some key points:

- The `byte`, `short`, `int`, and `long` types are used for integer values without decimal points.
- Each numeric type uses twice as many bits as the smaller similar type. For example, `short` uses twice as many bits as `byte` does.
- All of the numeric types are signed and reserve one of their bits to cover a negative range. For example, instead of `byte` covering 0 to 255 (or even 1 to 256), it actually covers -128 to 127.
- A `float` requires the letter `f` or `F` following the number so Java knows it is a float. Without an `f` or `F`, Java interprets a decimal value as a `double`.
- A `long` requires the letter `l` or `L` following the number so Java knows it is a `long`. Without an `l` or `L`, Java interprets a number without a decimal point as an `int` in most scenarios.

You won't be asked about the exact sizes of these types, although you should have a general idea of the size of smaller types like `byte` and `short`. A common question among newer Java developers is, what is the bit size of `boolean`? The answer is, it is not specified and is dependent on the JVM where the code is being executed.

Signed and Unsigned: *short* and *char*

For the exam, you should be aware that `short` and `char` are closely related, as both are stored as integral types with the same 16-bit length. The primary difference is that `short` is *signed*, which means it splits its range across the positive and negative integers. Alternatively, `char` is *unsigned*, which means its range is strictly positive, including 0.

Often, `short` and `char` values can be cast to one another because the underlying data size is the same. You learn more about casting in [Chapter 2](#), "Operators."

Writing Literals

There are a few more things you should know about numeric primitives. When a number is present in the code, it is called a *literal*. By default, Java assumes you are

defining an `int` value with a numeric literal. In the following example, the number listed is bigger than what fits in an `int`. Remember, you aren't expected to memorize the maximum value for an `int`. The exam will include it in the question if it comes up.

```
long max = 3123456789; // DOES NOT COMPILE
```

Java complains the number is out of range. And it is—for an `int`. However, we don't have an `int`. The solution is to add the character `L` to the number.

```
long max = 3123456789L; // Now Java knows it is a long
```

Alternatively, you could add a lowercase `l` to the number. But *please* use the uppercase `L`. The lowercase `l` looks like the number `1`.

Another way to specify numbers is to change the “base.” When you learned how to count, you studied the digits 0–9. This numbering system is called *base 10* since there are 10 possible values for each digit. It is also known as the *decimal number system*. Java allows you to specify digits in several other formats.

- Octal (digits 0–7), which uses the number `0` as a prefix—for example, `017`.
- Hexadecimal (digits 0–9 and letters A–F/a–f), which uses `0x` or `0X` as a prefix—for example, `0xFF`, `0xff`, `0XFf`. Hexadecimal is case insensitive, so all of these examples mean the same value.
- Binary (digits 0–1), which uses the number `0` followed by `b` or `B` as a prefix—for example, `0b10`, `0B10`.

Be sure to be able to recognize valid literal values that can be assigned to numbers.

Literals and the Underscore Character

The last thing you need to know about numeric literals is that you can have underscores in numbers to make them easier to read.

```
int million1 = 1000000;  
int million2 = 1_000_000;
```

We'd rather be reading the latter one because the zeros don't run together. You can add underscores anywhere except at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point. You can even place multiple underscore characters next to each other, although we don't recommend it.

Let's look at a few examples:

```
double notAtStart = _1000.00; // DOES NOT COMPILE  
double notAtEnd = 1000.00_; // DOES NOT COMPILE
```

```
double notByDecimal = 1000_.00;           // DOES NOT COMPILE
double annoyingButLegal = 1_00_0.0_0;    // Ugly, but compiles
double reallyUgly = 1 2;                  // Also compiles
```

Using Reference Types

A *reference type* refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to. Instead, a reference “points” to an object by storing the memory address where the object is located, a concept referred to as a *pointer*. Unlike other languages, Java does not allow you to learn what the physical memory address is. You can only use the reference to refer to the object.

Let’s take a look at some examples that declare and initialize reference types. Suppose we declare a reference of type `String`.

```
String greeting;
```

The `greeting` variable is a reference that can only point to a `String` object. A value is assigned to a reference in one of two ways.

- A reference can be assigned to another object of the same or compatible type.
- A reference can be assigned to a new object using the `new` keyword.

For example, the following statement assigns this reference to a new object:

```
greeting = new String("How are you?");
```


The `greeting` reference points to a new `String` object, "How are you?". The `String` object does not have a name and can be accessed only via a corresponding reference.

Objects vs. References

Do not confuse a reference with the object that it refers to; they are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. All references are the same size, no matter what their type is.

An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to

another object, and an object cannot be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.

 An image of two rectangular entities. On the left is a reference name connected to an object placed inside the heap.

Distinguishing Between Primitives and Reference Types

There are a few important differences you should know between primitives and reference types. First, notice that all the primitive types have lowercase type names. All classes that come with Java begin with uppercase. Although not required, it is a standard practice, and you should follow this convention for classes you create as well.

Next, reference types can be used to call methods, assuming the reference is not `null`. Primitives do not have methods declared on them. In this example, we can call a method on reference since it is of a reference type. You can tell `length` is a method because it has `()` after it. See if you can understand why the following snippet does not compile:

```
4: String reference = "hello";
5: int len = reference.length();
6: int bad = len.length(); // DOES NOT COMPILE
```

Line 6 is gibberish. No methods exist on `len` because it is an `int` primitive. Primitives do not have methods. Remember, a `String` is not a primitive, so you can call methods like `length()` on a `String` reference, as we did on line 5.

Finally, reference types can be assigned `null`, which means they do not currently refer to an object. Primitive types will give you a compiler error if you attempt to assign them `null`. In this example, `value` cannot point to `null` because it is of type `int`:

```
int value = null;    // DOES NOT COMPILE
String name = null;
```

But what if you don't know the value of an `int` and want to assign it to `null`? In that case, you should use a numeric wrapper class, such as `Integer`, instead of `int`.

Creating Wrapper Classes

Each primitive type has a wrapper class, which is an object type that corresponds to the primitive. [Table 1.6](#) lists all the wrapper classes along with how to create them.

TABLE 1.6 Wrapper classes

Primitive type	Wrapper class	Wrapper class inherits Number?	Example of creating
boolean	Boolean	No	<code>Boolean.valueOf(true)</code>
byte	Byte	Yes	<code>Byte.valueOf((byte) 1)</code>
short	Short	Yes	<code>Short.valueOf((short) 1)</code>
int	Integer	Yes	<code>Integer.valueOf(1)</code>
long	Long	Yes	<code>Long.valueOf(1)</code>
float	Float	Yes	<code>Float.valueOf((float) 1.0)</code>
double	Double	Yes	<code>Double.valueOf(1.0)</code>
char	Character	No	<code>Character.valueOf('c')</code>

Converting from *String* Using *valueOf()* Methods

The classes in [Table 1.6](#) include a `valueOf(String str)` method that converts a *String* into the associated wrapper class. For example:

```
int primitive = Integer.parseInt("123");
Integer wrapper = Integer.valueOf("123");
```

The first line converts a *String* to an `int` primitive. The second converts a *String* to an `Integer` wrapper class. On the numeric wrapper classes, `valueOf()` throws a `NumberFormatException` on invalid input. For example:

```
System.out.println(Integer.valueOf("five")); // NumberFormatException
```

On `Boolean`, the method returns `Boolean.TRUE` for any value that matches "true" ignoring case, and `Boolean.FALSE` otherwise. For example:

```
System.out.println(Boolean.valueOf("true")); // true
System.out.println(Boolean.valueOf("TrUe")); // true
System.out.println(Boolean.valueOf("false")); // false
System.out.println(Boolean.valueOf("FALSE")); // false
System.out.println(Boolean.valueOf("kangaroo")); // false
System.out.println(Boolean.valueOf(null)); // false
```

Finally, the numeric integral classes (`Byte`, `Short`, `Integer`, and `Long`) include an overloaded `valueOf(String str, int base)` method that takes a base value. As you

saw earlier, base 16, or hexadecimal includes the characters 0-9 along with A-Z. The overloaded `valueOf()` method allows you to pass any of these characters and ignores case. For example:

```
System.out.println(Integer.valueOf("5", 16)); // 5
System.out.println(Integer.valueOf("a", 16)); // 10
System.out.println(Integer.valueOf("F", 16)); // 15
System.out.println(Integer.valueOf("G", 16)); // NumberFormatException
```

This has been known to show up on exams from time to time, so make sure you understand these examples.

All of the numeric classes in [Table 1.6](#) extend the `Number` class, which means they all come with some useful helper methods: `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()`, and `doubleValue()`. The `Boolean` and `Character` wrapper classes include `booleanValue()` and `charValue()`, respectively.

As you probably guessed, these methods return the primitive value of a wrapper instance, in the type requested.

```
Double apple = Double.valueOf("200.99");
System.out.println(apple.byteValue()); // -56
System.out.println(apple.intValue()); // 200
System.out.println(apple.doubleValue()); // 200.99
```

These helper methods do their best to convert values but can result in a loss of precision. In the first example, there is no 200 in byte, so it wraps around to -56. In the second example, the value is truncated, which means all of the numbers after the decimal are dropped. In [Chapter 5](#), we apply autoboxing and unboxing to show how easy Java makes it to work with primitive and wrapper values.

Some of the wrapper classes contain additional helper methods for working with numbers. You don't need to memorize these methods; you can assume any you are given are valid. For example, `Integer` has the following:

- `max(int num1, int num2)`, which returns the largest of the two numbers
- `min(int num1, int num2)`, which returns the smallest of the two numbers
- `sum(int num1, int num2)`, which adds the two numbers

Defining Text Blocks

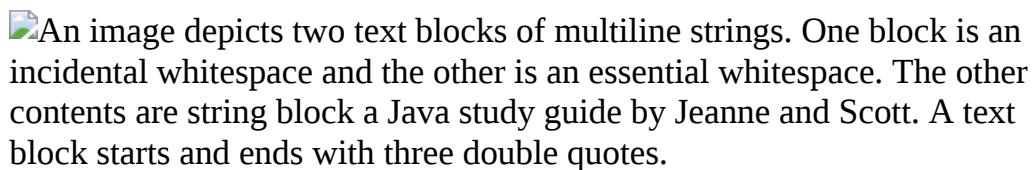
Earlier we saw a simple `String` with the value "hello". What if we want to have a `String` with something more complicated? For example, let's figure out how to create a `String` with this value:


```
"Java Study Guide"
  by Jeanne & Scott
```

Building this as a `String` requires two things you haven't learned yet. The syntax `\"` lets you say you want a `"` rather than to end the `String`, and `\n` says you want a new line. Both of these are called *escape characters* because the backslash provides a special meaning. With these two new skills, we can write this:

```
String eyeTest = "\"Java Study Guide\"\\n    by Jeanne & Scott";
```

While this does work, it is hard to read. Luckily, Java has *text blocks*, also known as multiline strings. See [Figure 1.3](#) for the text block equivalent.

An image depicts two text blocks of multiline strings. One block is an incidental whitespace and the other is an essential whitespace. The other contents are string block a Java study guide by Jeanne and Scott. A text block starts and ends with three double quotes.

[FIGURE 1.3](#) Text block

A text block starts and ends with three double quotes (`"""`), and the contents don't need to be escaped. This is much easier to read. Notice how the type is still `String`. This means any `String` methods you already know or will learn in [Chapter 4](#) apply to text blocks too. It also means you use a text block with a method that takes a `String`. For example:

```
public String label(String title, String author) {
    return """
        Book:
        """ + title + " by " + author;
}
public void prepare() {
    String labelled = label("""
        Java Study Guide
        For Java 21
        2024 Edition""", "Jeanne & Scott");
    System.out.println(labelled);
}
```

You might have noticed the words *incidental* and *essential whitespace* in the figure. What's that? *Essential whitespace* is part of your `String` and is important to you. *Incidental whitespace* just happens to be there to make the code easier to read. You can reformat your code and change the amount of incidental whitespace without any impact on your `String` value.

Imagine a vertical line drawn on the leftmost non-whitespace character in your text block. Everything to the left of it is incidental whitespace, and everything to the right

is essential whitespace. Let's try an example. How many lines does this output, and how many incidental and essential whitespace characters begin each line?

```
14: String pyramid = ""
15:   *
16:  * *
17: * * *
18: "";
19: System.out.print(pyramid);
```

There are four lines of output. Lines 15–17 have stars. Line 18 is a line without any characters. The closing triple " would have needed to be on line 17 if we didn't want that blank line. There are no incidental whitespace characters here. The closing "" on line 18 are the leftmost characters, so the line is drawn at the leftmost position. Line 15 has two essential whitespace characters to begin the line, and line 16 has one. That whitespace fills in the line drawn to match line 18.

[Table 1.7](#) shows some special formatting sequences and compares how they work in a regular String and a text block.

TABLE 1.7 Text block formatting

Formatting	Meaning in regular String	Meaning in text block
\ "	"	"
\ ""	n/a – Invalid	""
\ "\\"	""	""
Space (at end of line)	Space	Ignored
\s	Two spaces (\s is a space and preserves leading space on the line)	Two spaces
\ (at end of line)	n/a – Invalid	Omits new line on that line

Let's try a few examples. First, do you see why this doesn't compile?

```
String block = ""doe""; // DOES NOT COMPILE
```

Text blocks require a line break after the opening "", making this one invalid. Now let's try a valid one. How many lines do you think are in this text block?

```
String block = ""
doe \
deer"";
```

Just one. The output is doe deer since the \ tells Java not to add a new line before deer. Let's try determining the number of lines in another text block.

```
String block = ""  
    doe \n  
    deer  
    "";  
    ;
```

This time we have four lines. Since the text block has the closing "" on a separate line, we have three lines for the lines in the text block plus the explicit \n. Let's try one more. What do you think this outputs?

```
String block = ""  
    "doe\\\\"  
    "\\deer\\"  
    "";  
    ;  
System.out.println("'" + block + "'");
```

The answer is:

```
* "doe""  
  "deer""  
*
```

All of the \" escape the ". There is one space of essential whitespace on the doe and deer lines. All the other leading whitespace is incidental whitespace.

Declaring Variables

You've seen some variables already. A *variable* is a name for a piece of memory that stores data. When you declare a variable, you need to state the variable type along with giving it a name. Giving a variable a value is called *initializing* a variable. To initialize a variable, you just type the variable name followed by an equal sign, followed by the desired value. This example shows declaring and initializing a variable in one line:

```
String zooName = "The Best Zoo";
```

In the following sections, we look at how to properly define variables in one or multiple lines.

Identifying Identifiers

It probably comes as no surprise to you that Java has precise rules about identifier names. An *identifier* is the name of a variable, method, class, interface, or package.

Luckily, the rules for identifiers for variables apply to all of the other types that you are free to name.

There are only four rules to remember for legal identifiers.

- Identifiers must begin with a letter, a currency symbol, or a `_` symbol. Currency symbols include dollar (\$), yuan (¥), euro (€), and so on.
- Identifiers can include numbers but not start with them.
- A single underscore (`_`) is not allowed as an identifier.
- You cannot use the same name as a Java reserved word. A *reserved word* is a special word that Java has held aside so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that differ only in case. Please don't, though.

Don't worry—you won't need to memorize the full list of reserved words. The exam will only ask you about ones that are commonly used, such as `class` and `for`. [Table 1.8](#) lists all of the reserved words in Java.

TABLE 1.8 Reserved words

abstract	assert	boolean	break	byte
case	catch	char	class	const *
continue	default	do	double	else
enum	extends	final	finally	float
for	goto *	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

* The reserved words `const` and `goto` aren't actually used in Java. They are reserved so that people coming from other programming languages don't use them by accident—and, in theory, in case Java wants to use them one day.

There are other names you can't use. For example, `true`, `false`, and `null` are literal values, so they can't be variable names. Additionally, there are contextual keywords like `module` in [Chapter 12](#). Prepare to be tested on these rules. The following examples are legal:

```
long okidentifier;  
float $OK2Identifier;  
boolean _alsoOK1d3ntifi3r;  
char __SStill0kbutKnotsonice$;
```

These examples are not legal:

```
int 3DPointClass;    // identifiers cannot begin with a number  
byte hollywood@vine; // @ is not a letter, digit, $ or _  
String *$coffee;    // first character * is not a letter, $ or _  
double public;       // public is a reserved word  
short _;             // a single underscore is not allowed
```

camelCase and snake_case

Although you can do crazy things with identifier names, please don't. Java has conventions so that code is readable and consistent. For example, *camelCase* has the first letter of each word capitalized. Method and variable names are typically written in camelCase with the first letter lowercase, such as `toUpperCase()`. Class and interface names are also written in camelCase, with the first letter uppercase, such as `ArrayList`.

Another style is called *snake_case*. It simply uses an underscore (`_`) to separate words. Java generally uses uppercase snake_case for constants and enum values, such as `NUMBER_FLAGS`.

The exam will not always follow these conventions to make questions about identifiers trickier. By contrast, questions on other topics generally do follow standard conventions. We recommend you follow these conventions on the job.

Declaring Multiple Variables

You can also declare and initialize multiple variables in the same statement. How many variables do you think are declared and initialized in the following example?

```
void sandFence() {  
    String s1, s2;  
    String s3 = "yes", s4 = "no";  
}
```

Four String variables were declared: `s1`, `s2`, `s3`, and `s4`. You can declare many variables in the same declaration as long as they are all of the same type. You can also initialize any or all of those values inline. In the previous example, we have two

initialized variables: s3 and s4. The other two variables remain declared but not yet initialized.

This is where it gets tricky. Pay attention to tricky things! The exam will attempt to trick you. Again, how many variables do you think are declared and initialized in the following code?

```
void paintFence() {  
    int i1, i2, i3 = 0;  
}
```

As you should expect, three variables were declared: i1, i2, and i3. However, only one of those values was initialized: i3. The other two remain declared but not yet initialized. That's the trick. Each snippet separated by a comma is a little declaration of its own. The initialization of i3 only applies to i3. It doesn't have anything to do with i1 or i2 despite being in the same statement. As you will see in the next section, you can't actually use i1 or i2 until they have been initialized.

Another way the exam could try to trick you is to show you code like this line:

```
int num, String value; // DOES NOT COMPILE
```

This code doesn't compile because it tries to declare multiple variables of *different* types in the same statement. The shortcut to declare multiple variables in the same statement is legal only when they share a type.



Legal, *valid*, and *compiles* are all synonyms in the Java exam world. We try to use all the terminology you could encounter on the exam.

To make sure you understand this, see if you can figure out which of the following are legal declarations:

```
4: boolean b1, b2;  
5: String s1 = "1", s2;  
6: double d1, double d2;  
7: int i1; int i2;  
8: int i3; i4;
```

Lines 4 and 5 are legal. They each declare two variables. Line 4 doesn't initialize either variable, and line 5 initializes only one. Line 7 is also legal. Although int does

appear twice, each one is in a separate statement. A semicolon (;) separates statements in Java. It just so happens there are two completely different statements on the same line.

Line 6 is *not* legal. Java does not allow you to declare two different types in the same statement. Wait a minute! Variables `d1` and `d2` are the same type. They are both of type `double`. Although that's true, it still isn't allowed. If you want to declare multiple variables in the same statement, they must share the same type declaration and not repeat it.

Line 8 is *not* legal. Again, we have two completely different statements on the same line. The second one on line 8 is not a valid declaration because it omits the type. When you see an oddly placed semicolon on the exam, pretend the code is on separate lines and think about whether the code compiles that way. In this case, the last two lines of code could be rewritten as follows:

```
int i1;  
int i2;  
int i3;  
i4;
```

Looking at the last line on its own, you can easily see that the declaration is invalid. And yes, the exam really does cram multiple statements onto the same line—partly to try to trick you and partly to fit more code on the screen. In the real world, please limit yourself to one declaration per statement and line. Your teammates will thank you for the readable code.

Initializing Variables

Before you can use a variable, it needs a value. Some types of variables get this value set automatically, and others require the programmer to specify it. In the following sections, we look at the differences between the defaults for local, instance, and class variables.

Creating Local Variables

A *local variable* is a variable defined within a constructor, method, or initializer block. For simplicity, we focus primarily on local variables within methods in this section, although the rules for the others are the same.

Final Local Variables

The `final` keyword can be applied to local variables and is equivalent to declaring constants in other languages. Consider this example:

```
5: final int y = 10;
6: int x = 20;
7: y = x + 10;    // DOES NOT COMPILE
```

Both variables are set, but `y` uses the `final` keyword. For this reason, line 7 triggers a compiler error since the value cannot be modified.

The `final` modifier can also be applied to local variable references. The following example uses an `int[]` array object, which you learn about in [Chapter 4](#).

```
5: final int[] favoriteNumbers = new int[10];
6: favoriteNumbers[0] = 10;
7: favoriteNumbers[1] = 20;
8: favoriteNumbers = null;    // DOES NOT COMPILE
```

Notice that we can modify the content, or data, in the array. The compiler error isn't until line 8, when we try to change the value of the reference `favoriteNumbers`.

Uninitialized Local Variables

Local variables do not have a default value and must be initialized before use. Furthermore, the compiler will report an error if you try to read an uninitialized value. For example, the following code generates a compiler error:

```
4: public int notValid() {
5:     int y = 10;
6:     int x;
7:     int reply = x + y;    // DOES NOT COMPILE
8:     return reply;
9: }
```

The `y` variable is initialized to 10. By contrast, `x` is not initialized before it is used in the expression on line 7, and the compiler generates an error. The compiler is smart enough to recognize variables that have been initialized after their declaration but before they are used. Here's an example:

```
public int valid() {
    int y = 10;
    int x;    // x is declared here
    x = 3;    // x is initialized here
    int z;    // z is declared here but never initialized or used
    int reply = x + y;
    return reply;
}
```


In this example, `x` is declared, initialized, and used in separate lines. Also, `z` is declared but never used, so it is not required to be initialized.

The compiler is also smart enough to recognize initializations that are more complex. In this example, there are two branches of code.

```
public void findAnswer(boolean check) {
    int answer;
    int otherAnswer;
    int onlyOneBranch;
    if (check) {
        onlyOneBranch = 1;
        answer = 1;
    } else {
        answer = 2;
    }
    System.out.println(answer);
    System.out.println(onlyOneBranch); // DOES NOT COMPILE
}
```

The `answer` variable is initialized in both branches of the `if` statement, so the compiler is perfectly happy. It knows that regardless of whether `check` is `true` or `false`, the value `answer` will be set to something before it is used. The `otherAnswer` variable is not initialized but never used, and the compiler is equally as happy. Remember, the compiler is concerned only if you try to use uninitialized local variables; it doesn't mind the ones you never use.

The `onlyOneBranch` variable is initialized only if `check` happens to be `true`. The compiler knows there is the possibility for `check` to be `false`, resulting in uninitialized code, and gives a compiler error. You learn more about the `if` statement in [Chapter 3](#), “Making Decisions.”



On the exam, be wary of any local variable that is declared but not initialized in a single line. This is a common place on the exam that could result in a “Does not compile” answer. Be sure to check to make sure it's initialized before it's used on the exam.

Passing Constructor and Method Parameters

Variables passed to a constructor or method are called *constructor parameters* or *method parameters*, respectively. These parameters are like local variables that have been pre-initialized. The rules for initializing constructor and method parameters are the same, so we focus primarily on method parameters.

In the previous example, `check` is a method parameter.

```
public void findAnswer(boolean check) {}
```

Take a look at the following method `checkAnswer()` in the same class:

```
public void checkAnswer() {  
    boolean value;  
    findAnswer(value); // DOES NOT COMPILE  
}
```

The call to `findAnswer()` does not compile because it tries to use a variable that is not initialized. While the caller of a method `checkAnswer()` needs to be concerned about the variable being initialized, once inside the method `findAnswer()`, we can assume the local variable has been initialized to some value.

Defining Instance and Class Variables

Variables that are not local variables are defined either as instance variables or as class variables. An *instance variable*, often called a field, is a value defined within a specific instance of an object. Let's say we have a `Person` class with an instance variable `name` of type `String`. Each instance of the class would have its own value for `name`, such as `Elysia` or `Sarah`. Two instances could have the same value for `name`, but changing the value for one does not modify the other.

On the other hand, a *class variable* is one that is defined on the class level and shared among all instances of the class. It can even be publicly accessible to classes outside the class and doesn't require an instance to use. In our previous `Person` example, a shared class variable could be used to represent the list of people at the zoo today. You can tell a variable is a class variable because it has the keyword `static` before it. You learn about this in [Chapter 5](#). For now, just know that a variable is a class variable if it has the `static` keyword in its declaration.

Instance and class variables do not require you to initialize them. As soon as you declare these variables, they are given a default value. The compiler doesn't know what value to use and so wants the simplest value it can give the type: `null` for an object, `zero` for the numeric types, and `false` for a `boolean`. You don't need to know the default value for `char`, but in case you are curious, it is `'\u0000'` (NUL).

Inferring the Type with *var*

You have the option of using the keyword *var* instead of the type when declaring local variables under certain conditions. To use this feature, you just type *var* instead of the primitive or reference type. Here's an example:

```
public class Zoo {  
    public void whatTypeAmI() {  
        var name = "Hello";  
        var size = 7;  
    }  
}
```

The formal name of this feature is *local variable type inference*. Let's take that apart. First comes *local variable*. This means just what it sounds like. You can use this feature only for local variables. The exam may try to trick you with code like this:

```
public class VarKeyword {  
    var tricky = "Hello"; // DOES NOT COMPILE  
}
```

Wait a minute! We just learned the difference between instance and local variables. The variable *tricky* is an instance variable. Local variable type inference works with local variables and not instance variables.

Type Inference of *var*

Now that you understand the local variable part, it is time to go on to what *type inference* means. The good news is that this also means what it sounds like. When you type *var*, you are instructing the compiler to determine the type for you. The compiler looks at the code on the line of the declaration and uses it to infer the type. Take a look at this example:

```
7: public void reassignment() {  
8:     var number = 7;  
9:     number = 4;  
10:    number = "five"; // DOES NOT COMPILE  
11: }
```

On line 8, the compiler determines that we want an *int* variable. On line 9, we have no trouble assigning a different *int* to it. On line 10, Java has a problem. We've asked it to assign a *String* to an *int* variable. This is not allowed. It is equivalent to typing this:

```
int number = "five";
```



If you know a language like JavaScript, you might be expecting `var` to mean a variable that can take on any type at runtime. In Java, `var` is still a specific type defined at compile time. It does not change type at runtime.

For simplicity when discussing `var`, we are going to assume a variable declaration statement is completed in a single line. You could insert a line break between the variable name and its initialization value, as in the following example:

```
7: public void breakingDeclaration() {  
8:     var silly  
9:         = 1;  
10: }
```

This example is valid and does compile, but we consider the declaration and initialization of `silly` to be happening on the same line.

Examples with `var`

Let's go through some more scenarios so the exam doesn't trick you on this topic! Do you think the following compiles?

```
3: public void doesThisCompile(boolean check) {  
4:     var question;  
5:     question = 1;  
6:     var answer;  
7:     if (check) {  
8:         answer = 2;  
9:     } else {  
10:        answer = 3;  
11:    }  
12:    System.out.println(answer);  
13: }
```

The code does not compile. Remember that for local variable type inference, the compiler looks only at the line with the declaration. Since `question` and `answer` are not assigned values on the lines where they are defined, the compiler does not know what to make of them. For this reason, both lines 4 and 6 do not compile.

You might find that strange since both branches of the `if/else` do assign a value. Alas, it is not on the same line as the declaration, so it does not count for `var`. Contrast

this behavior with what we saw a short while ago when we discussed branching and initializing a local variable in our `findAnswer()` method.

Now we know the initial value used to determine the type needs to be part of the same statement. Can you figure out why these two statements don't compile?

```
4: public void twoTypes() {  
5:     int a, var b = 3;    // DOES NOT COMPILE  
6:     var a, b = 3;       // DOES NOT COMPILE  
7:     var n = null;       // DOES NOT COMPILE  
8: }
```

Line 5 wouldn't work even if you replaced `var` with a real type. All the types declared on a single line must be the same type and share the same declaration. We couldn't write `int a, int b = 3;` either. Line 6 shows that you can't use `var` to define two variables on the same line.

Line 7 is a single line. The compiler is being asked to infer the type of `null`. This could be any reference type. The only choice the compiler could make is `Object`. However, that is almost certainly not what the author of the code intended. The designers of Java decided it would be better not to allow `var` for `null` than to have to guess at intent.



While a `var` cannot be initialized with a `null` value without a type, it can be reassigned a `null` value after it is declared, provided that the underlying data type is a reference type.

Let's try another example. Do you see why this does not compile?

```
public int addition(var a, var b) { // DOES NOT COMPILE  
    return a + b;  
}
```

In this example, `a` and `b` are method parameters. These are not local variables. Be on the lookout for `var` used with constructor parameters, method parameters, or instance variables. Using `var` in one of these places is a good exam trick to see if you are paying attention. Remember that `var` is used only for local variable type inference!

There's one last rule you should be aware of: `var` is not a reserved word and allowed to be used as an identifier. It is considered a reserved type name. A *reserved type name* means it cannot be used to define a type, such as a class, interface, or enum. Do you think this is legal?

```
package var;

public class Var {
    public void var() {
        var var = "var";
    }
    public void Var() {
        Var var = new Var();
    }
}
```

Believe it or not, this code does compile. Java is case sensitive, so `Var` doesn't introduce any conflicts as a class name. Naming a local variable `var` is legal. Please don't write code that looks like this at your job! But understanding why it works will help get you ready for any tricky exam questions the exam creators could throw at you.



Real World Scenario

***var* in the Real World**

The `var` keyword is great for exam authors because it makes it easier to write tricky code. When you work on a real project, you want the code to be easy to read.

Once you start having code that looks like the following, it is time to consider using `var`:

```
PileOfPapersToFileInFilingCabinet pileOfPapersToFile =
    new PileOfPapersToFileInFilingCabinet();
```

You can see how shortening this would be an improvement without losing any information:

```
var pileOfPapersToFile = new PileOfPapersToFileInFilingCabinet();
```

If you are ever unsure whether it is appropriate to use `var`, we recommend “Local Variable Type Inference: Style Guidelines,” which is available at the following location:

Managing Variable Scope

You've learned that local variables are declared within a code block. How many variables do you see that are scoped to this method?

```
public void eat(int piecesOfCheese) {  
    int bitesOfCheese = 1;  
}
```

There are two variables with local scope. The `bitesOfCheese` variable is declared inside the method. The `piecesOfCheese` variable is a method parameter. Neither variable can be used outside of where it is defined.

Limiting Scope

Local variables can never have a scope larger than the method they are defined in. However, they can have a smaller scope. Consider this example:

```
3: public void eatIfHungry(boolean hungry) {  
4:     if (hungry) {  
5:         int bitesOfCheese = 1;  
6:     } // bitesOfCheese goes out of scope here  
7:     System.out.println(bitesOfCheese); // DOES NOT COMPILE  
8: }
```

The variable `hungry` has a scope of the entire method, while the variable `bitesOfCheese` has a smaller scope. It is only available for use in the `if` statement because it is declared inside of it. When you see a set of braces (`{}`) in the code, it means you have entered a new block of code. Each block of code has its own scope. When there are multiple blocks, you match them from the inside out. In our case, the `if` statement block begins at line 4 and ends at line 6. The method's block begins at line 3 and ends at line 8.

Since `bitesOfCheese` is declared in an `if` statement block, the scope is limited to that block. When the compiler gets to line 7, it complains that it doesn't know anything about this `bitesOfCheese` thing and gives an error.

Remember that blocks can contain other blocks. These smaller contained blocks can reference variables defined in the larger scoped blocks, but not vice versa. Here's an example:

```

16: public void eatIfHungry(boolean hungry) {
17:     if (hungry) {
18:         int bitesOfCheese = 1;
19:         {
20:             var teenyBit = true;
21:             System.out.println(bitesOfCheese);
22:         }
23:     }
24:     System.out.println(teenyBit); // DOES NOT COMPILE
25: }

```

The variable defined on line 18 is in scope until the block ends on line 23. Using it in the smaller block from lines 19 to 22 is fine. The variable defined on line 20 goes out of scope on line 22. Using it on line 24 is not allowed.

Tracing Scope

The exam will attempt to trick you with various questions on scope. You'll probably see a question that appears to be about something complex and fails to compile because one of the variables is out of scope.

Let's try one. Don't worry if you aren't familiar with `if` statements or `while` loops yet. It doesn't matter what the code does since we are talking about scope. See if you can figure out on which line each of the five local variables goes into and out of scope.

```

11: public void eatMore(boolean hungry, int amountOfFood) {
12:     int roomInBelly = 5;
13:     if (hungry) {
14:         var timeToEat = true;
15:         while (amountOfFood > 0) {
16:             int amountEaten = 2;
17:             roomInBelly = roomInBelly - amountEaten;
18:             amountOfFood = amountOfFood - amountEaten;
19:         }
20:     }
21:     System.out.println(amountOfFood);
22: }

```

This method does compile. The first step in figuring out the scope is to identify the blocks of code. In this case, there are three blocks. You can tell this because there are three sets of braces. Starting from the innermost set, we can see where the `while` loop's block starts and ends. Repeat this process as we go on for the `if` statement block and method block. [Table 1.9](#) shows the line numbers that each block starts and ends on.

TABLE 1.9 Tracking scope by block

Line	First line in block	Last line in block
while	15	19
if	13	20
Method	11	22

Now that we know where the blocks are, we can look at the scope of each variable. `hungry` and `amountOfFood` are method parameters, so they are available for the entire method. This means their scope is lines 11 to 22. The variable `roomInBelly` goes into scope on line 12 because that is where it is declared. It stays in scope for the rest of the method and goes out of scope on line 22. The variable `timeToEat` goes into scope on line 14 where it is declared. It goes out of scope on line 20 where the `if` block ends. Finally, the variable `amountEaten` goes into scope on line 16 where it is declared. It goes out of scope on line 19 where the `while` block ends.

You'll want to practice this skill a lot! Identifying blocks and variable scope needs to be second nature for the exam. The good news is that there are lots of code examples to practice on. You can look at any code example on any topic in this book and match up braces.

Applying Scope to Classes

All of that was for local variables. Luckily, the rule for instance variables is easier: they are available as soon as they are defined and last for the entire lifetime of the object itself. The rule for class, aka `static`, variables is even easier: they go into scope when declared like the other variable types. However, they stay in scope for the entire life of the program.

Let's do one more example to make sure you have a handle on this. Again, try to figure out the type of the four variables and when they go into and out of scope.

```

1: public class Mouse {
2:     final static int MAX_LENGTH = 5;
3:     int length;
4:     public void grow(int inches) {
5:         if (length < MAX_LENGTH) {
6:             int newSize = length + inches;
7:             length = newSize;
8:         }
9:     }
10: }
```

In this class, we have one class variable, `MAX_LENGTH`; one instance variable, `length`; and two local variables, `inches` and `newSize`. The `MAX_LENGTH` variable is a class

variable because it has the `static` keyword in its declaration. In this case, `MAX_LENGTH` goes into scope on line 2 where it is declared. It stays in scope until the program ends.

Next, `length` goes into scope on line 3 where it is declared. It stays in scope as long as this `Mouse` object exists. `inches` goes into scope where it is declared on line 4. It goes out of scope at the end of the method on line 9. `newSize` goes into scope where it is declared on line 6. Since it is defined inside the `if` statement block, it goes out of scope when that block ends on line 8.

Reviewing Scope

Got all that? Let's review the rules on scope.

- *Local variables*: In scope from declaration to the end of the block
- *Method parameters*: In scope for the duration of the method
- *Instance variables*: In scope from declaration until the object is eligible for garbage collection
- *Class variables*: In scope from declaration until the program ends

Not sure what garbage collection is? Relax, that's our next and final section for this chapter.

Destroying Objects

Now that we've played with our objects, it is time to put them away. Luckily, the JVM takes care of that for you. Java provides a garbage collector to automatically look for objects that aren't needed anymore.

Remember, your code isn't the only process running in your Java program. Java code exists inside of a JVM, which includes numerous processes independent from your application code. One of the most important of those is a built-in garbage collector.

All Java objects are stored in your program memory's *heap*. The heap, which is also referred to as the *free store*, represents a large pool of unused memory allocated to your Java application. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory and crash. Oh, no! Luckily, garbage collection solves this problem. In the following sections, we look at garbage collection.

Understanding Garbage Collection

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program. There are many different algorithms for garbage collection, but you don't need to know any of them for the exam.

As a developer, the most interesting part of garbage collection is determining when the memory belonging to an object can be reclaimed. In Java and other languages, *eligible for garbage collection* refers to an object's state of no longer being accessible in a program and therefore able to be garbage collected.

Does this mean an object that's eligible for garbage collection will be immediately garbage collected? Definitely not. When the object actually is discarded is not under your control, but for the exam, you will need to know at any given moment which objects are eligible for garbage collection.

Think of garbage-collection eligibility like shipping a package. You can take an item, seal it in a labeled box, and put it in your mailbox. This is analogous to making an item eligible for garbage collection. When the mail carrier comes by to pick it up, though, is not in your control. For example, it may be a postal holiday, or there could be a severe weather event. You can even call the post office and ask them to come pick it up right away, but there's no way to guarantee when and if this will actually happen. Ideally, they will come by before your mailbox fills with packages!

Java includes a built-in method to help support garbage collection where you can suggest that garbage collection run.

```
System.gc();
```

Just like the post office, Java is free to ignore you. This method is not *guaranteed* to do anything.

Tracing Eligibility

How does the JVM know when an object is eligible for garbage collection? The JVM waits patiently and monitors each object until it determines that the code no longer needs that memory. An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of these two situations occurs:

- The object no longer has any references pointing to it.
- All references to the object have gone out of scope.

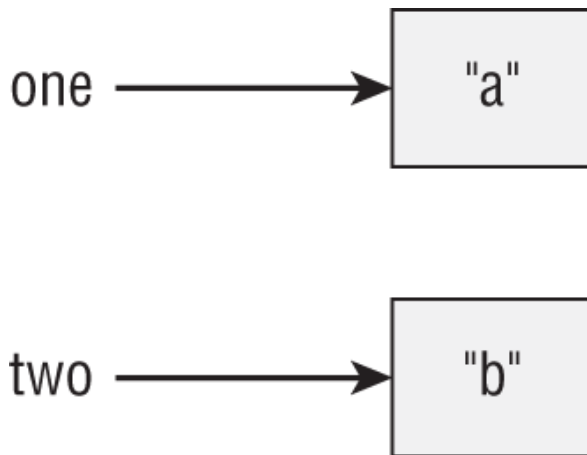
Realizing the difference between a reference and an object goes a long way toward understanding garbage collection, the `new` operator, and many other facets of the Java

language. Look at this code and see whether you can figure out when each object first becomes eligible for garbage collection:

```
1: public class Scope {  
2:     public static void main(String[] args) {  
3:         String one, two;  
4:         one = new String("a");  
5:         two = new String("b");  
6:         one = two;  
7:         String three = one;  
8:         one = null;  
9:     } }
```

When you are asked a question about garbage collection on the exam, we recommend that you draw what's going on. There's a lot to keep track of in your head, and it's easy to make a silly mistake trying to hold it all in your memory. Let's try it together now. Really. Get a pencil and paper. We'll wait.

Got that paper? OK, let's get started. On line 3, write **one** and **two** (just the words—no need for boxes or arrows since no objects have gone on the heap yet). On line 4, we have our first object. Draw a box with the string **"a"** in it, and draw an arrow from the word **one** to that box. Line 5 is similar. Draw another box with the string **"b"** in it this time and an arrow from the word **two**. At this point, your work should look like [Figure 1.4](#).



[FIGURE 1.4](#) Your drawing after line 5

On line 6, the variable **one** changes to point to **"b"**. Either erase or cross out the arrow from **one** and draw a new arrow from **one** to **"b"**. On line 7, we have a new variable, so write the word **three** and draw an arrow from **three** to **"b"**. Notice that **three** points to what **one** is pointing to right now and not what it was pointing to at the beginning. This is why you are drawing pictures. It's easy to forget something like that. At this point, your work should look like [Figure 1.5](#).

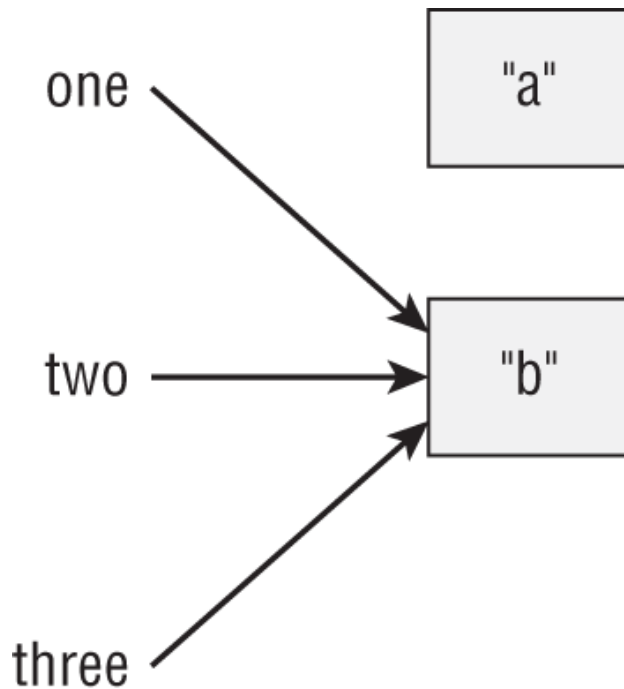


FIGURE 1.5 Your drawing after line 7

Finally, cross out the line between one and "b" since line 8 sets this variable to null. Now, we were trying to find out when the objects were first eligible for garbage collection. On line 6, we got rid of the only arrow pointing to "a", making that object eligible for garbage collection. "b" has arrows pointing to it until it goes out of scope. This means "b" doesn't go out of scope until the end of the method on line 9.

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

You'll see code that doesn't have a 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. Finally, remember that extra whitespace doesn't matter in Java syntax. The exam may use varying amounts of whitespace to trick you.

Summary

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.

Primitive types are the basic building blocks of Java types. They are assembled into reference types. Reference types can have methods and be assigned a `null` value. Numeric literals are allowed to contain underscores (`_`) as long as they do not start or end the literal and are not next to a decimal point (`.`). Wrapper classes are reference types, and there is one for each primitive. Text blocks allow creating a `String` on multiple lines using `"""`.

Declaring a variable involves stating the data type and giving the variable a name. Variables that represent fields in a class are automatically initialized to their corresponding `0`, `null`, or `false` values during object instantiation. Local variables must be specifically initialized before they can be used. Identifiers may contain letters, numbers, currency symbols, or `_`. Identifiers may not begin with numbers. Local variable declarations may use the `var` keyword instead of the actual type. When using `var`, the type is set once at compile time and does not change.

Scope refers to that portion of code where a variable can be accessed. There are three kinds of variables in Java, depending on their scope: instance variables, class variables, and local variables. Instance variables are the non-static fields of your class. Class variables are the static fields within a class. Local variables are declared within a constructor, method, or initializer block.