Chapter 2
Java Building Blocks

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **Working With Java Primitive Data Types and String APIs**
  - Declare and initialize variables (including casting and promoting primitive data types)
  - Identify the scope of variables
  - Use local variable type inference
- **Describing and Using Objects and Classes**
  - Declare and instantiate Java objects, and explain objects' lifecycles (including creation, dereferencing by reassignment, and garbage collection)
  - Read or write to object fields

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'll be presenting the basic structure of Java 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 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.

Creating Objects

Our programs wouldn't be able to do anything useful if we didn't have the ability to create new objects. Remember from Chapter 1, "Welcome to Java," that an object is an instance of a class. In the following sections, we'll 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'll likely 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 declare a constructor. You'll learn all about them in Chapter 8, "Class Design."

Some classes provide built-in methods that allow you to create new instances without using a constructor or the `new` keyword. For example, in Chapter 5, "Core Java APIs," you'll create instances of `Integer` using the `valueOf()` method. Methods like this will often use `new` with a constructor in their method definition. For the exam, remember that anytime a constructor is used, the `new` keyword is required.

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. Reading a variable is known as *getting* it. The class gets `numberEggs` directly to print it out. Writing to a variable is known as *setting* it. This class sets `numberEggs` to 1.

In Chapter 7, "Methods and Encapsulation," you'll learn how to use encapsulation to protect the `Swan` class from having someone set a negative number of eggs.

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 7, you'll 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 of 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 only called when the `main()` method is executed. There is one additional set of braces on lines 1 and 6 that constitute the class declaration.

Following 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'll add some more rules to the order of initialization in Chapter 8. In the meantime, you need to remember:

- 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 ini-

tializer 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 will see a lot more of rules and examples covering order of initialization in Chapter 8.

Understanding Data Types

Java applications contain two types of data: primitive types and reference types. In this section, we'll 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 2.1 shows the Java primitive types together with their size in bits and the range of values that each holds.

TABLE 2.1 Primitive types

Keyword	Type	Example
`boolean`	true or false	`true`
`byte`	8-bit integral value	`123`
`short`	16-bit integral value	`123`
`int`	32-bit integral value	`123`
`long`	64-bit integral value	`123L`
`float`	32-bit floating-point value	`123.45f`
`double`	64-bit floating-point value	`123.456`
`char`	16-bit Unicode value	`'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'll learn more about `String` in Chapter 5, but for now just remember they are objects, not primitives.

There's a lot of information in Table 2.1. Let's look at some key points:

- The `float` and `double` types are used for floating-point (decimal) values.
- A `float` requires the letter `f` following the number so Java knows it is a float.

- The `byte`, `short`, `int`, and `long` types are used for numbers without decimal points. In mathematics, these are all referred to as integral values, but in Java, `int` and `Integer` refer to specific types.
- 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 in Java. This means that they reserve one of their bits to cover a negative range. For example, `byte` ranges from `-128` to `127`. You might be surprised that the range is not `-128` to `128`. Don't forget, `0` needs to be accounted for too in the range.

You won't be asked about the exact sizes of most of these types, although you should know that a `byte` can hold a value from `–128` to `127`.

Real World Scenario

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 range is strictly positive including `0`. Therefore, `char` can hold a higher positive numeric value than `short`, but cannot hold any negative numbers.

The compiler allows them to be used interchangeably in some cases, as shown here:

```
short bird = 'd';
char mammal = (short)83;
```

Printing each variable displays the value associated with their type.

```
System.out.println(bird);   // Prints 100
System.out.println(mammal); // Prints S
```

This usage is not without restriction, though. If you try to set a value outside the range of `short` or `char`, the compiler will report an error.

```
short reptile = 65535;  // DOES NOT COMPILE
char fish = (short)-1;  // DOES NOT COMPILE
```

Both of these examples would compile if their data types were swapped because the values would then be within range for their type. You'll learn more about casting in Chapter 3, "Operators."

So you aren't stuck memorizing data type ranges, let's look at how Java derives it from the number of bits. A `byte` is 8 bits. A bit has two possible values. (These are basic computer science definitions that you should memorize.) 2^8 is 2 × 2 = 4 × 2 = 8 × 2 = 16 × 2 = 32 × 2 = 64 × 2 = 128 × 2 = 256. Since `0` needs to be included in the range, Java takes it away from the positive side. Or if you don't like math, you can just memorize it.

FLOATING-POINT NUMBERS AND SCIENTIFIC NOTATION

While integer values like `short` and `int` are relatively easy to calculate the range for, floating-point values like `double` and `float` are decidedly not. In most computer systems, floating-point numbers are stored in scientific notation. This means the numbers are stored as two numbers, `a` and `b`, of the form `a x 10` $^{\texttt{b}}$.

This notation allows much larger values to be stored, at the cost of accuracy. For example, you can store a value of `3 x 10` $^{\texttt{200}}$ in a `double`, which would require a lot more than 8 bytes if every digit were stored without scientific notation (84 bytes in case you were wondering). To accomplish this, you only store the first dozen or so digits of the number. The name *scientific notation* comes from science, where often only the first few significant digits are required for a calculation.

Don't worry, for the exam you are not required to know scientific notation or how floating-point values are stored.

The number of bits is used by Java when it figures out how much memory to reserve for your variable. For example, Java allocates 32 bits if you write this:

```
int num;
```

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 numbers. 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`

You won't need to convert between number systems on the exam. You'll have 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 `java.util.Date` and a reference of type `String`:

```
java.util.Date today;
String greeting;
```

The `today` variable is a reference of type `Date` and can only point to a `Date` object. 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 statements assign these references to new objects:

```
today = new java.util.Date();
greeting = new String("How are you?");
```

The `today` reference now points to a new `Date` object in memory, and `today` can be used to access the various fields and methods of this `Date` object. Similarly, the `greeting` reference points to a new `String` object, `"How are you?"`. The `String` and `Date` objects do not have names and can be accessed only via their corresponding reference. Figure 2.1 shows how the reference types appear in memory.

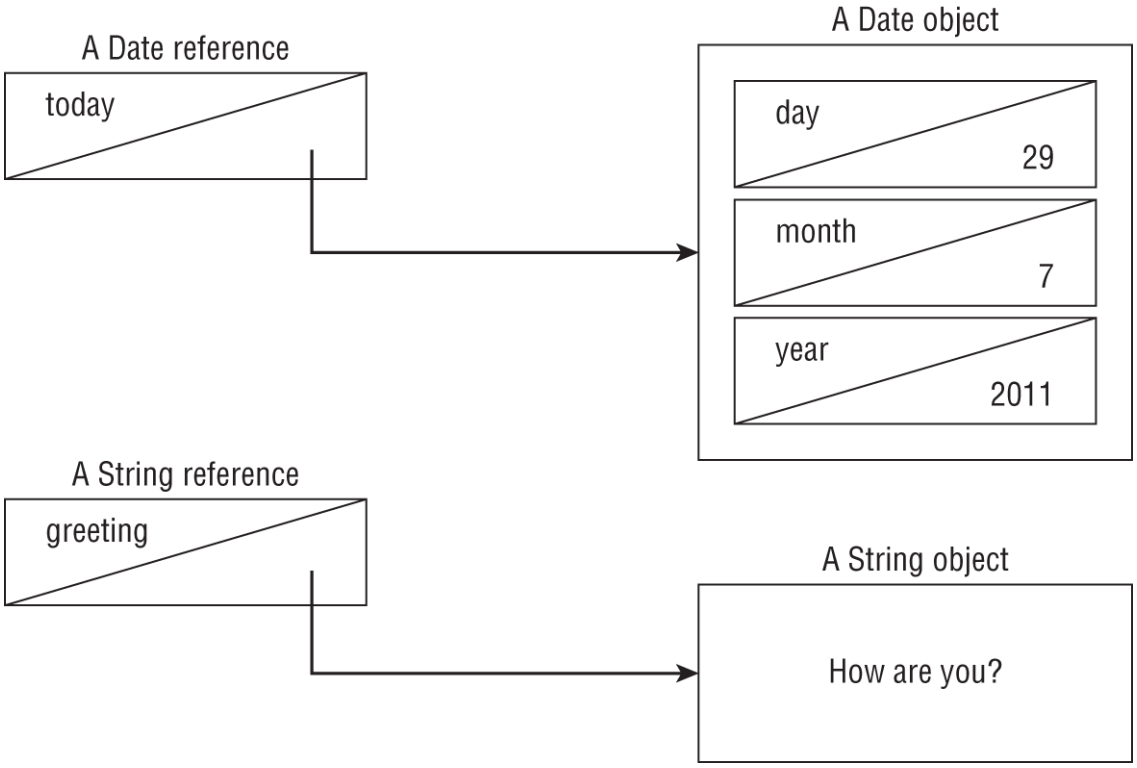


FIGURE 2.1 An object in memory can be accessed only via a reference.

Distinguishing between Primitives and Reference Types

There are a few important differences you should know between primitives and reference types. First, 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 s = 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`. Wrapper classes will be covered in Chapter 5.

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, 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.

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. For example, the following code declares two variables. One is named `zooName` and is of type `String`. The other is named `numberAnimals` and is of type `int`.

```
String zooName;
int numberAnimals;
```

Now that we've declared a variable, we can give it a value. This 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:

```
zooName = "The Best Zoo";
numberAnimals = 100;
```

Since you often want to initialize a variable right away, you can do so in the same statement as the declaration. For example, here we merge the previous declarations and initializations into more concise code:

```
String zooName = "The Best Zoo";
int numberAnimals = 100;
```

In the following sections, we'll 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 `$` symbol, or a `_` symbol.
- Identifiers can include numbers but not start with them.
- Since Java 9, a single underscore `_` is not allowed as an identifier.
- You cannot use the same name as a Java reserved word. A *reserved word* is 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 only differ 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 2.2 lists all of the reserved words in Java.

TABLE 2.2 Reserved words

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

* 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.

** `true` / `false` / `null` are not actually reserved words, but literal values. Since they cannot be used as identifier names, we include them in this table.

Prepare to be tested on these rules. The following examples are legal:

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

These examples are not legal:

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

Style: camelCase

Although you can do crazy things with identifier names, please don't. Java has conventions so that code is readable and consistent. This consistency includes *camel case*, often written as camelCase for emphasis. In camelCase, the first letter of each word is capitalized.

The camelCase format makes identifiers easier to read. Which would you rather read: `Thisismyclass name` or `ThisIsMyClass name` ? The exam will mostly use common conventions for identifiers, but not always. When you see a nonstandard identifier, be sure to check if it is legal. If it's not, you get to mark the answer "does not compile" and skip analyzing everything else in the question.

IDENTIFIERS IN THE REAL WORLD

Most Java developers follow these conventions for identifier names:

- Method and variable names are written in camelCase with the first letter being lowercase.
- Class and interface names are written in camelCase with the first letter being uppercase. Also, don't start any class name with `$`, as the compiler uses this symbol for some files.

Also, know that valid letters in Java are not just characters in the English alphabet. Java supports the Unicode character set, so there are thousands of characters that can start a legal Java identifier. Some are non-Arabic numerals that may appear after the first character in a legal identifier. Luckily, you don't have to worry about memorizing those for the exam. If you are in a country that doesn't use the English alphabet, this is useful to know for a job.

Style: snake_case

Another style you might see both on the exam and in the real world or in other languages is called *snake case*, often written as snake_case for emphasis. It simply uses an underscore (_) to separate words, often entirely in lowercase. The previous example would be written as `this_is_my_class name` in snake_case.

While both camelCase and snake_case are perfectly valid syntax in Java, the development community functions better when everyone adopts the same style convention. With that in mind, Oracle (and Sun before it) recommends everyone use camelCase for class and variable names. There are some exceptions, though. Constant `static final` values are often written in snake_case, such as `THIS_IS_A_CONSTANT`. In addition, `enum` values tend to be written with snake_case, as in `Color.RED`, `Color.DARK_GRAY`, and so on.

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;
```

The first statement on line 4 is legal. It declares two variables without initializing them. The second statement on line 5 is also legal. It declares two variables and initializes only one of them.

The third statement on 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. `double d1, d2;` would have been legal.

The fourth statement on line 7 is 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. The fifth statement on 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'll 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 will focus primarily on local variables within methods in this section, although the rules for the others are the same.

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`. However, because `x` is not initialized before it is used in the expression on line 7, the compiler generates the following error:

```
Test.java:7: variable x might not have been initialized
   int reply = x + y; // DOES NOT COMPILE
               ^
```

Until `x` is assigned a value, it cannot appear within an expression, and the compiler will gladly remind you of this rule. The compiler knows your code has control of what happens inside the method and can be expected to initialize values.

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; // and initialized here
   int reply = x + y;
   return reply;
}
```

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 only concerned 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'll learn more about the `if` statement in Chapter 4, "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. As you saw in the previous examples, you are not required to initialize the variable on the same line it is defined, but 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 local variables that have been pre-initialized. In other words, they are like local variables that have been initialized before the method is called, by the caller. The rules for initializing constructor and method parameters are the same, so we'll 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 without requiring 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'll learn about this in Chapter 7. 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. You'll need to memorize everything in Table 2.3 except the default value of `char`. To

make this easier, remember that the compiler doesn't know what value to use and so wants the simplest value it can give the type: `null` for an object and `0`/`false` for a primitive.

TABLE 2.3 Default initialization values by type

Variable type	Default initialization value
`boolean`	`false`
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`char`	`'\u0000'` (NUL)
All object references (everything else)	`null`

Introducing *var*

Starting in Java 10, you have the option of using the keyword `var` instead of the type for 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 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 only use this feature 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.

In Chapter 4, you'll learn that `var` can be used in `for` loops, and as you'll see in Chapter 6, "Lambdas and Functional Interfaces," with some lambdas as well. In Chapter 10, "Exceptions," you'll also learn that `var` can be used with try-with-resources. All of these cases are still internal to a method and therefore consistent with what you learn in this chapter.

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.

So, the type of `var` can't change at runtime, but what about the value? Take a look at the following code snippet:

```
var apples = (short)10;
apples = (byte)5;
apples = 1_000_000;  // DOES NOT COMPILE
```

The first line creates a `var` named `apples` with a type of `short`. It then assigns a `byte` of `5` to it, but did that change the data type of `apples` to `byte`? Nope! As you will learn in Chapter 3, the `byte` can be automatically promoted to a `short`, because a `byte` is small enough that it can fit inside of `short`. Therefore, the value stored on the second line is a `short`. In fact, let's rewrite the example showing what the compiler is really doing when it sees the `var`:

```
short apples = (short)10;
apples = (byte)5;
apples = 1_000_000;  // DOES NOT COMPILE
```

The last line does not compile, as one million is well beyond the limits of `short`. The compiler treats the value as an `int` and reports an error indicating it cannot be assigned to `apples`.

If you didn't follow that last example, don't worry, we'll be covering numeric promotion and casting in the next chapter. For now, you just need to know that the value for a `var` can change after it is declared but the type never does.

For simplicity when discussing `var` in the following sections, we are going to assume a variable declaration statement is completed in a single line. For example, 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 n = null;      // DOES NOT COMPILE
7: }
```

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 v = 3;` either. Likewise, this is not allowed:

```
5:    var a = 2, b = 3;  // DOES NOT COMPILE
```

In other words, Java does not allow `var` in multiple variable declarations.

Line 6 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.

VAR* AND *NULL

While a `var` cannot be initialized with a `null` value without a type, it can be assigned a `null` value after it is declared, provided that the underlying data type of the `var` is an object. Take a look at the following code snippet:

```
13: var n = "myData";
14: n = null;
15: var m = 4;
16: m = null;  // DOES NOT COMPILE
```

Line 14 compiles without issue because `n` is of type `String`, which is an object. On the other hand, line 16 does not compile since the type of `m` is a primitive `int`, which cannot be assigned a `null` value.

It might surprise you to learn that a `var` can be initialized to a `null` value if the type is specified. You'll learn about casting in Chapter 3, but the following does compile:

```
17: var o = (String)null;
```

Since the type is provided, the compiler can apply type inference and set the type of the `var` to be `String`.

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 constructors, 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 only used for local variable type inference!

Time for two more examples. 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 Oracle could throw at you!

There's one last rule you should be aware of. While `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`. For example, the following code snippet does not compile because of the class name:

```
public class var {  // DOES NOT COMPILE
   public var() {
   }
}
```

We're sure if the writers of Java had a time machine, they would likely go back and make `var` a reserved word in Java 1.0. They could have made `var` a reserved word starting in Java 10 or 11, but this would have broken older code where `var` was used as a variable name. For a large enough project, making `var` a reserved word could involve checking and recompiling millions of lines of code! On the other hand, since having a class or interface start with a lowercase letter is considered a bad practice prior to Java 11, they felt pretty safe marking it as a reserved type name.

It is often inappropriate to use `var` as the type for every local variable in your code. That just makes the code difficult to understand. If you are ever unsure of whether it is appropriate to use `var`, there are numerous style guides out there that can help. We recommend the one titled "Style Guidelines for Local Variable Type Inference in Java," which is available at the following location. This resource includes great style suggestions.

https://openjdk.java.net/projects/amber/LVTIstyle.html

Review of *var* Rules

We complete this section by summarizing all of the various rules for using `var` in your code. Here's a quick review of the `var` rules:

1. A `var` is used as a local variable in a constructor, method, or initializer block.
2. A `var` cannot be used in constructor parameters, method parameters, instance variables, or class variables.
3. A `var` is always initialized on the same line (or statement) where it is declared.
4. The value of a `var` can change, but the type cannot.
5. A `var` cannot be initialized with a `null` value without a type.
6. A `var` is not permitted in a multiple-variable declaration.
7. A `var` is a reserved type name but not a reserved word, meaning it can be used as an identifier except as a class, interface, or `enum` name.

That's a lot of rules, but we hope most are pretty straightforward. Since `var` is new to Java since the last exam, expect to see it used frequently on the exam. You'll also be seeing numerous ways `var` can be used throughout this book.

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();
```

Managing Variable Scope

You've learned that local variables are declared within a method. How many local variables do you see in this example?

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

There are two local variables in this method. The `bitesOfCheese` variable is declared inside the method. The `piecesOfCheese` variable is a method parameter and, as discussed earlier, it also acts like a local variable in terms of garbage collection and scope. Both of these variables are said to have a *scope* local to the method. This means they cannot be used outside of where they are 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 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:

```
error: cannot find symbol
   System.out.println(bitesOfCheese);  // DOES NOT COMPILE
                      ^
   symbol:   variable bitesOfCheese
```

Nesting Scope

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 de-

fined 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: }
```

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 as we go out for the `if` statement block and method block. Table 2.4 shows the line numbers that each block starts and ends on.

TABLE 2.4 **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 so 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 end of block
- *Instance variables*: In scope from declaration until object eligible for garbage collection
- *Class variables*: In scope from declaration until 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 automatically takes care of that for you. Java provides a garbage collector to automatically look for objects that aren't needed anymore.

Remember from Chapter 1, your code isn't the only process running in your Java program. Java code exists inside of a Java Virtual Machine (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. The heap may be quite large, depending on your environment, but there is always a limit to its size. After all, there's no such thing as a computer with infinite memory. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory and crash.

In the following sections, we'll look at garbage collection.

Real World Scenario

GARBAGE COLLECTION IN OTHER LANGUAGES

One of the distinguishing characteristics of Java since its very first version is that it automatically performs garbage collection for you. In fact, other than removing references to an object, there's very little you can do to control garbage collection directly in Java.

While garbage collection is pretty standard in most programming languages now, some languages, such as C, do not have automatic garbage collection. When a developer finishes using an object in memory, they have to manually deallocate it so the memory can be reclaimed and reused.

Failure to properly handle garbage collection can lead to catastrophic performance and security problems, the most common of which is for an application to run out of memory. Another similar problem, though, is if secure data like a credit card number stays in memory long after it is used and is able to be read by other programs. Luckily, Java handles a lot of these complex issues for you.

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. If you are curious, though, one algorithm is to keep a counter on the number of places an object is accessible at any given time and mark it eligible for garbage collection if the counter ever reaches zero.

Eligible for Garbage Collection

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. Hopefully, they come by before your mailbox fills with packages!

As a programmer, the most important thing you can do to limit out-of-memory problems is to make sure objects are eligible for garbage collection once they are no longer needed. It is the JVM's responsibility to actually perform the garbage collection.

Calling *System.gc()*

Java includes a built-in method to help support garbage collection that can be called at any time.

```
public static void main(String[] args) {
   System.gc();
}
```

What is the `System.gc()` command *guaranteed* to do? Nothing, actually. It merely *suggests* that the JVM kick off garbage collection. The JVM may perform garbage collection at that moment, or it might be busy and choose not to. The JVM is free to ignore the request.

When is `System.gc()` *guaranteed* to be called by the JVM? Never, actually. While the JVM will likely run it over time as available memory decreases, it is not guaranteed to ever actually run. In fact, shortly before a program runs out of memory and throws an `OutOfMemoryError`, the JVM will *try* to perform garbage collection, but it's not guaranteed to succeed.

For the exam, you need to know that `System.gc()` is not guaranteed to run or do anything, and you should be able to recognize when objects become eligible for garbage collection.

Tracing Eligibility

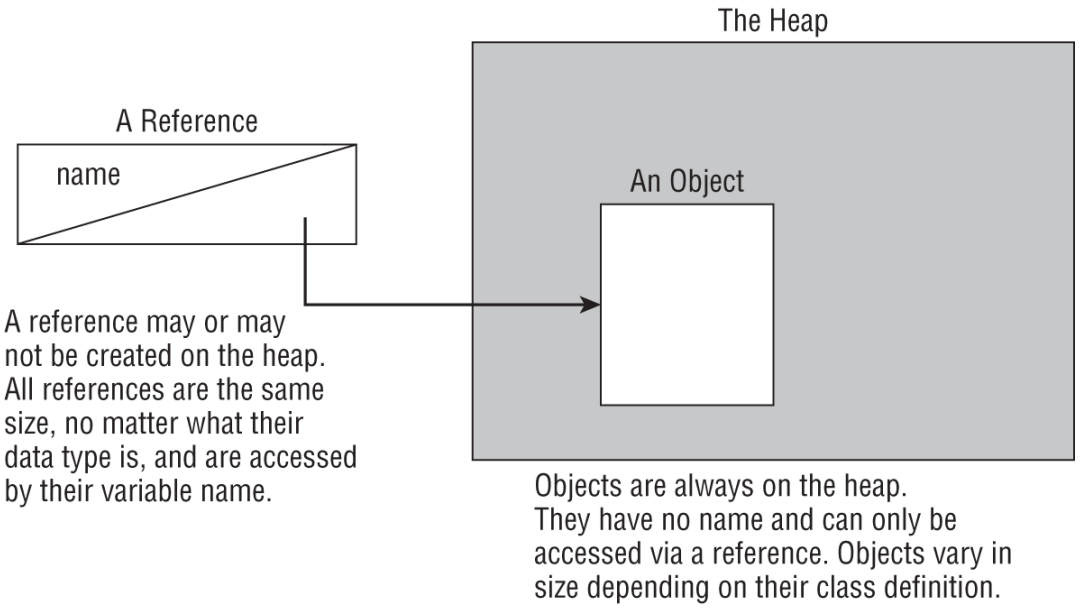
How does the JVM know when an object is eligible for garbage collection? The JVM waits patiently and monitors each object until it determines that the code no longer needs that memory. An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of two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

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

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.



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 get asked a question about garbage collection on the exam, we recommend 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 keep it all in your memory. Let's try it together now. Really. Get a pencil and paper. We'll wait.

Got that paper? Okay, let's get started. On line 3, write **`one`** and **`two`** (just the words—no need for boxes or arrows yet 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 2.2.

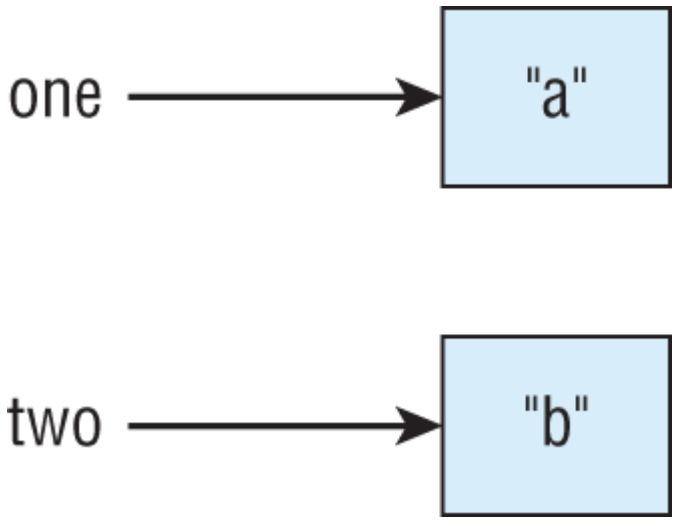


FIGURE 2.2 Your drawing after line 5

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 2.3.

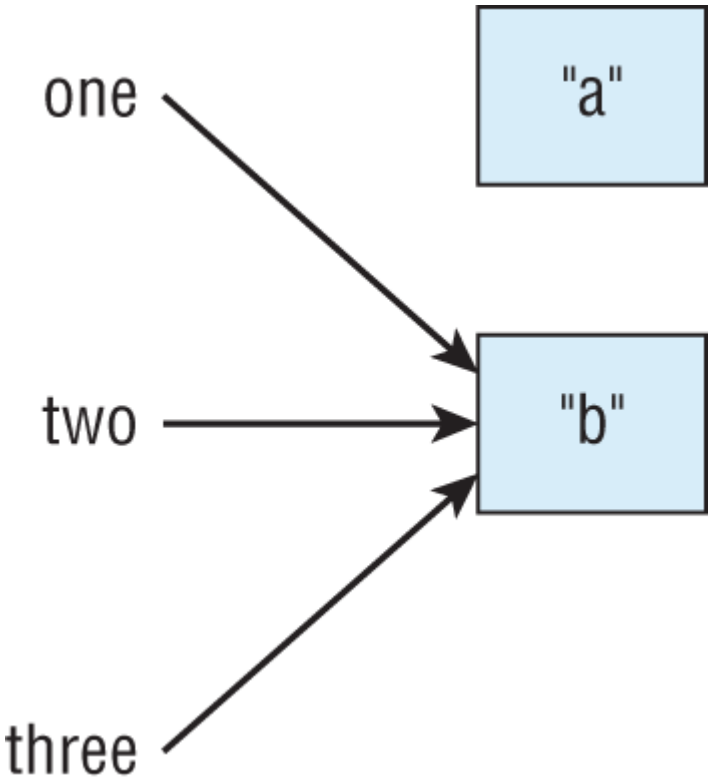


FIGURE 2.3 Your drawing after line 7

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.

FINALIZE()

Java allows objects to implement a method called `finalize()`. This feature can be confusing and hard to use properly. In a nutshell, the garbage collector would call the `finalize()` method once. If the garbage collector didn't run, there was no call to `finalize()`. If the garbage collector failed to collect the object and tried again later, there was no second call to `finalize()`.

This topic is no longer on the exam. In fact, it is deprecated in `Object` as of Java 9, with the official documentation stating, "The finalization mechanism is inherently problematic." We mention the `finalize()` method in case Oracle happens to borrow from an old exam question. Just remember that `finalize()` can run zero or one times. It cannot run twice.

Summary

In this chapter, we described the building blocks of Java—most important, what a Java object is, how it is referenced and used, and how it is destroyed. This chapter lays the foundation for many topics that we will revisit throughout this book.

For example, we will go into a lot more detail on primitive types and how to use them in Chapter 3. Creating methods will be covered in Chapter 7. And in Chapter 8, we will discuss numerous rules for creating and managing objects. In other words, learn the basics, but don't worry if you didn't follow everything in this chapter. We will go a lot deeper into many of these topics in the rest of the book.

To begin with, constructors create Java objects. A constructor is a method matching the class name and omitting the return type. When an object is instantiated, fields and blocks of code are initialized first. Then the constructor is run.

Next, primitive types are the basic building blocks of Java types. They are assembled into reference types. Reference types can have methods and be assigned to `null`. 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 (`.`).

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, `$`, or `_`. Identifiers may not begin with numbers. Local variables 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.

Moving on, 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.

Finally, garbage collection is responsible for removing objects from memory when they can never be used again. An object becomes eligible for garbage collection when there are no more references to it or its references have all gone out of scope.

Exam Essentials

Be able to recognize a constructor. A constructor has the same name as the class. It looks like a method without a return type.

Be able to identify legal and illegal declarations and initialization. Multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization;

others use the default value for that type. Identifiers may contain letters, numbers, `$`, or `_`, although they may not begin with numbers. Also, you cannot define an identifier that is just a single underscore character `_`. Numeric literals may contain underscores between two digits, such as `1_000`, but not in other places, such as `_100_.0_`. Numeric literals can begin with `1–9`, `0`, `0x`, `0X`, `0b`, and `0B`, with the latter four indicating a change of numeric base.

Be able to use `var` correctly. A `var` is used for a local variable inside a constructor, a method, or an initializer block. It cannot be used for constructor parameters, method parameters, instance variables, or class variables. A `var` is initialized on the same line where it is declared, and while it can change value, it cannot change type. A `var` cannot be initialized with a `null` value without a type, nor can it be used in multiple variable declarations. Finally, `var` is not a reserved word in Java and can be used as a variable name.

Be able to determine where variables go into and out of scope. All variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is eligible for garbage collection. Class variables remain in scope as long as the program is running.

Know how to identify when an object is eligible for garbage collection. Draw a diagram to keep track of references and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

Review Questions

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

1. Which of the following are valid Java identifiers? (Choose all that apply.)

1. `_`
2. `_helloWorld$`
3. `true`
4. `java.lang`
5. `Public`
6. `1980_s`
7. `_Q2_`

2. What lines are printed by the following program? (Choose all that apply.)

```
1: public class WaterBottle {
2:    private String brand;
3:    private boolean empty;
4:    public static float code;
5:    public static void main(String[] args) {
6:      WaterBottle wb = new WaterBottle();
7:      System.out.println("Empty = " + wb.empty);
8:      System.out.println("Brand = " + wb.brand);
9:      System.out.println("Code = " + code);
10:   } }
```

1. Line 8 generates a compiler error.
2. Line 9 generates a compiler error.
3. `Empty =`
4. `Empty = false`
5. `Brand =`
6. `Brand = null`
7. `Code = 0.0`
8. `Code = 0f`

3. Which of the following code snippets about `var` compile without issue when used in a method? (Choose all that apply.)
   1. `var spring = null;`
   2. `var fall = "leaves";`
   3. `var evening = 2; evening = null;`
   4. `var night = new Object();`

5. `var day = 1/0;`
6. `var winter = 12, cold;`
7. `var fall = 2, autumn = 2;`
8. `var morning = ""; morning = null;`

4. Which of the following statements about the code snippet are true? (Choose all that apply.)
    1. `4: short numPets = 5L;`
    2. `5: int numGrains = 2.0;`
    3. `6: String name = "Scruffy";`
    4. `7: int d = numPets.length();`
    5. `8: int e = numGrains.length;`
    6. `9: int f = name.length();`
    1. Line 4 generates a compiler error.
    2. Line 5 generates a compiler error.
    3. Line 6 generates a compiler error.
    4. Line 7 generates a compiler error.
    5. Line 8 generates a compiler error.
    6. Line 9 generates a compiler error.
5. Which statements about the following class are true? (Choose all that apply.)

```
1: public class River {
2:    int Depth = 1;
3:    float temp = 50.0;
4:    public void flow() {
5:       for (int i = 0; i < 1; i++) {
6:          int depth = 2;
7:          depth++;
8:          temp--;
9:       }
10:      System.out.println(depth);
11:      System.out.println(temp); }
12:   public static void main(String... s) {
13:      new River().flow();
14: } }
```

    1. Line 3 generates a compiler error.
    2. Line 6 generates a compiler error.
    3. Line 7 generates a compiler error.
    4. Line 10 generates a compiler error.
    5. The program prints `3` on line 10.
    6. The program prints `4` on line 10.
    7. The program prints `50.0` on line 11.
    8. The program prints `49.0` on line 11.
6. Which of the following are correct? (Choose all that apply.)
    1. An instance variable of type `float` defaults to `0`.
    2. An instance variable of type `char` defaults to `null`.
    3. An instance variable of type `double` defaults to `0.0`.
    4. An instance variable of type `int` defaults to `null`.
    5. An instance variable of type `String` defaults to `null`.
    6. An instance variable of type `String` defaults to the empty string `""`.
    7. None of the above
7. Which of the following are correct? (Choose all that apply.)
    1. A local variable of type `boolean` defaults to `null`.
    2. A local variable of type `float` defaults to `0.0f`.
    3. A local variable of type `double` defaults to `0`.
    4. A local variable of type `Object` defaults to `null`.
    5. A local variable of type `boolean` defaults to `false`.
    6. A local variable of type `float` defaults to `0.0`.
    7. None of the above
8. Which of the following are true? (Choose all that apply.)
    1. A class variable of type `boolean` defaults to `0`.
    2. A class variable of type `boolean` defaults to `false`.
    3. A class variable of type `boolean` defaults to `null`.
    4. A class variable of type `long` defaults to `null`.
    5. A class variable of type `long` defaults to `0L`.
    6. A class variable of type `long` defaults to `0`.
    7. None of the above
9. Which of the following statements about garbage collection are correct? (Choose all that apply.)

1. Calling `System.gc()` is guaranteed to free up memory by destroying objects eligible for garbage collection.
2. Garbage collection runs on a set schedule.
3. Garbage collection allows the JVM to reclaim memory for other objects.
4. Garbage collection runs when your program has used up half the available memory.
5. An object may be eligible for garbage collection but never removed from the heap.
6. An object is eligible for garbage collection once no references to it are accessible in the program.
7. Marking a variable `final` means its associated object will never be garbage collected.

10. Which statements about the following class are correct? (Choose all that apply.)

```
1: public class PoliceBox {
2:    String color;
3:    long age;
4:    public void PoliceBox() {
5:       color = "blue";
6:       age = 1200;
7:    }
8:    public static void main(String []time) {
9:       var p = new PoliceBox();
10:      var q = new PoliceBox();
11:      p.color = "green";
12:      p.age = 1400;
13:      p = q;
14:      System.out.println("Q1="+q.color);
15:      System.out.println("Q2="+q.age);
16:      System.out.println("P1="+p.color);
17:      System.out.println("P2="+p.age);
18: } }
```

1. It prints `Q1=blue`.
2. It prints `Q2=1200`.

3. It prints `P1=null`.
4. It prints `P2=1400`.
5. Line 4 does not compile.
6. Line 12 does not compile.
7. Line 13 does not compile.
8. None of the above

11. Which of the following legally fill in the blank so you can run the `main()` method from the command line? (Choose all that apply.)

```
public static void main(_________________) {}
```

1. `String... var`
2. `String My.Names[]`
3. `String[] 123`
4. `String[] _names`
5. `String... $n`
6. `var names`
7. `String myArgs`

12. Which of the following expressions, when inserted independently into the blank line, allow the code to compile? (Choose all that apply.)

```
public void printMagicData(__________) {
   double magic = ;
   System.out.println(magic);
}
```

1. `3_1`
2. `1_329_.0`
3. `3_13.0_`
4. `5_291._2`
5. `2_234.0_0`
6. `9___6`
7. `_1_3_5_0`
8. None of the above

13. Suppose we have a class named `Rabbit`. Which of the following statements are true? (Choose all that apply.)

```
1: public class Rabbit {
2:    public static void main(String[] args) {
3:       Rabbit one = new Rabbit();
4:       Rabbit two = new Rabbit();
5:       Rabbit three = one;
6:       one = null;
7:       Rabbit four = one;
8:       three = null;
9:       two = null;
10:      two = new Rabbit();
11:      System.gc();
12: } }
```

1. The `Rabbit` object created on line 3 is first eligible for garbage collection immediately following line 6.
2. The `Rabbit` object created on line 3 is first eligible for garbage collection immediately following line 8.
3. The `Rabbit` object created on line 3 is first eligible for garbage collection immediately following line 12.
4. The `Rabbit` object created on line 4 is first eligible for garbage collection immediately following line 9.
5. The `Rabbit` object created on line 4 is first eligible for garbage collection immediately following line 11.
6. The `Rabbit` object created on line 4 is first eligible for garbage collection immediately following line 12.
7. The `Rabbit` object created on line 10 is first eligible for garbage collection immediately following line 11.
8. The `Rabbit` object created on line 10 is first eligible for garbage collection immediately following line 12.

14. Which of the following statements about `var` are true? (Choose all that apply.)
    1. A `var` can be used as a constructor parameter.
    2. The type of `var` is known at compile time.
    3. A `var` cannot be used as an instance variable.
    4. A `var` can be used in a multiple variable assignment statement.
    5. The value of `var` cannot change at runtime.

6. The type of `var` cannot change at runtime.
7. The word `var` is a reserved word in Java.

15. Given the following class, which of the following lines of code can independently replace `INSERT CODE HERE` to make the code compile? (Choose all that apply.)

```
public class Price {
   public void admission() {
      INSERT CODE HERE
      System.out.print(amount);
      } }
```

1. `int Amount = 0b11;`
2. `int amount = 9L;`
3. `int amount = 0xE;`
4. `int amount = 1_2.0;`
5. `double amount = 1_0_.0;`
6. `int amount = 0b101;`
7. `double amount = 9_2.1_2;`
8. `double amount = 1_2_.0_0;`

16. Which statements about the following class are correct? (Choose all that apply.)

```
1: public class ClownFish {
2:    int gills = 0, double weight=2;
3:    { int fins = gills; }
4:    void print(int length = 3) {
5:       System.out.println(gills);
6:       System.out.println(weight);
7:       System.out.println(fins);
8:       System.out.println(length);
9: } }
```

1. Line 2 contains a compiler error.
2. Line 3 contains a compiler error.

3. Line 4 contains a compiler error.
4. Line 7 contains a compiler error.
5. The code prints `0`.
6. The code prints `2.0`.
7. The code prints `2`.
8. The code prints `3`.

17. Which statements about classes and its members are correct? (Choose all that apply.)
    1. A variable declared in a loop cannot be referenced outside the loop.
    2. A variable cannot be declared in an instance initializer block.
    3. A constructor argument is in scope for the life of the instance of the class for which it is defined.
    4. An instance method can only access instance variables declared before the instance method declaration.
    5. A variable can be declared in an instance initializer block but cannot be referenced outside the block.
    6. A constructor can access all instance variables.
    7. An instance method can access all instance variables.
18. Which statements about the following code snippet are correct? (Choose all that apply.)

```
3: var squirrel = new Object();
4: int capybara = 2, mouse, beaver = -1;
5: char chipmunk = -1;
6: squirrel = "";
7: beaver = capybara;
8: System.out.println(capybara);
9: System.out.println(mouse);
10: System.out.println(beaver);
11: System.out.println(chipmunk);
```

    1. The code prints `2`.
    2. The code prints `-1`.
    3. The code prints the empty `String`.
    4. The code prints: `null`.

5. Line 4 contains a compiler error.
6. Line 5 contains a compiler error.
7. Line 9 contains a compiler error.
8. Line 10 contains a compiler error.

19. Assuming the following class compiles, how many variables defined in the class or method are in scope on the line marked `// SCOPE` on line 14?

```
1: public class Camel {
2:    { int hairs = 3_000_0; }
3:    long water, air=2;
4:    boolean twoHumps = true;
5:    public void spit(float distance) {
6:       var path = "";
7:       { double teeth = 32 + distance++; }
8:       while(water > 0) {
9:          int age = twoHumps ? 1 : 2;
10:         short i=-1;
11:         for(i=0; i<10; i++) {
12:            var Private = 2;
13:         }
14:         // SCOPE
15:      }
16:   }
17: }
```

1. 2
2. 3
3. 4
4. 5
5. 6
6. 7
7. None of the above

20. What is the output of executing the following class?

```
1: public class Salmon {
2:    int count;
3:    { System.out.print(count+"-"); }
4:    { count++; }
5:    public Salmon() {
6:       count = 4;
7:       System.out.print(2+"-");
8:    }
9:    public static void main(String[] args) {
10:      System.out.print(7+"-");
11:      var s = new Salmon();
12:      System.out.print(s.count+"-"); } }
```

1. `7-0-2-1-`
2. `7-0-1-`
3. `0-7-2-1-`
4. `7-0-2-4-`
5. `0-7-1-`
6. The class does not compile because of line 3.
7. The class does not compile because of line 4.
8. None of the above.

21. Which statements about the following program are correct? (Choose all that apply.)

```
1: public class Bear {
2:    private Bear pandaBear;
3:    protected void finalize() {}
4:    private void roar(Bear b) {
5:       System.out.println("Roar!");
6:       pandaBear = b;
7:    }
8:    public static void main(String[] args) {
9:       Bear brownBear = new Bear();
10:      Bear polarBear = new Bear();
11:      brownBear.roar(polarBear);
12:      polarBear = null;
```

```
13:      brownBear = null;
14:      System.gc(); } }
```

1. The object created on line 9 is eligible for garbage collection after line 13.
2. The object created on line 9 is eligible for garbage collection after line 14.
3. The object created on line 10 is eligible for garbage collection after line 12.
4. The object created on line 10 is eligible for garbage collection after line 13.
5. Garbage collection is guaranteed to run.
6. Garbage collection might or might not run.
7. Garbage collection is guaranteed not to run.
8. The code does not compile.

22. Which of the following are valid instance variable declarations? (Choose all that apply.)
    1. `var _ = 6000_.0;`
    2. `var null = 6_000;`
    3. `var $_ = 6_000;`
    4. `var $2 = 6_000f;`
    5. `var var = 3_0_00.0;`
    6. `var #CONS = 2_000.0;`
    7. `var %C = 6_000_L;`
    8. None of the above