OCA(1Z0-808) JAVA SE 8 PROGRAMMER I QUICK REFERENCE GUIDE

Lakshmanakumar Kalisetti

Table of Contents

Java Building Blocks

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

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

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

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.

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. In addition to "normal" numbers, numeric literals are allowed to begin with 0 (octal), 0x (hex), 0X (hex), 0b (binary), or 0B (binary). Numeric literals are also allowed to contain underscores as long as they are directly between two other numbers.

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 "zero" or null value during object instantiation. Local variables must be specifically initialized. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers.

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 method.

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

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. The finalize() method will run once for each object if/when it is first garbage collected.

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

In the event of a conflict, class name imports take precedence.

Numeric literals may contain underscores between two digits and begin with 1–9, 0, 0x, 0X, 0b, and 0B.

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 garbage collected. Class variables remain in scope as long as the program is running.

Java allows objects to implement a method called finalize() that might get called. This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn't run, the method doesn't get called. If the garbage collector fails to collect the object and tries to run it again later, the method doesn't get called a second time.

You can even put two classes in the same file. When you do so, at most one of the classes in the file is allowed to be public. If you do have a public class, it needs to match the filename. public class Animal2 would not compile in a file named Animal.java

To compile Java code, the fi le must have the extension .java. The name of the fi le must match the name of the class. The result is a fi le of *bytecode* by the same name, but with a .class filename extension. Bytecode consists of instructions that the JVM knows how to execute. Notice that we must omit the .class extension to run Zoo.java because the period has a reserved meaning in the JVM.

While executing a java class from command line, If you want spaces inside an argument, you need to use double quotes

When the class is found in multiple packages, Java gives you the compiler error. If you explicitly import a class name, it takes precedence over any wildcards present. Sometimes you really do want to use a class with the same name from two different packages. When this happens, you can pick one to use in the import and use the other's fully qualified class name.

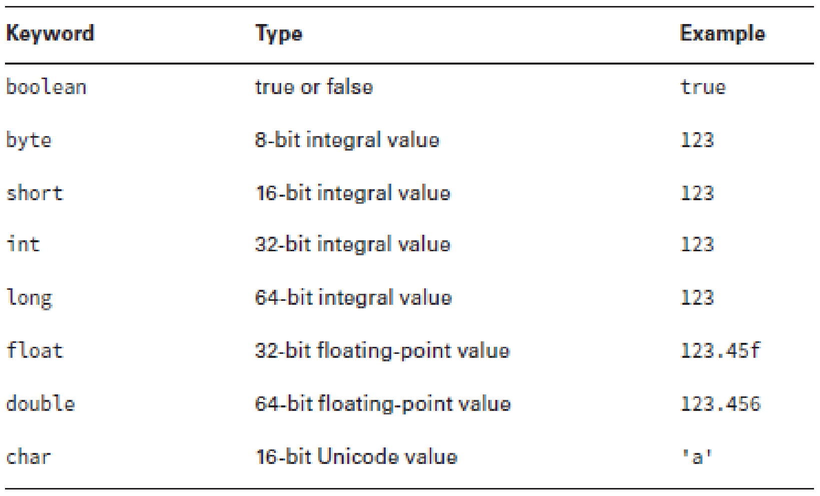
The purpose of a constructor is to initialize fields, although you can put any code in there.

You can even read and write fields directly on the line declaring them.

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*. 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.

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.

TABLE 1.1 Java 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	`123`
`float`	32-bit floating-point value	`123.45f`
`double`	64-bit floating-point value	`123.456`
`char`	16-bit Unicode value	`'a'`

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 literal.
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.

Binary, Octal and Hex numeric literals works with only integers, but not with double/float. We cannot have decimal places in case of these literals. For Hex & Binary, will get compilation errors. Octal representation will be treated as regular floating point number by ignoring the starting point 0(zero).

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. Next, reference types can be used to call methods when they do not point to null. Primitives do not have methods declared on them. Finally, notice that all the primitive types have lowercase type names. All classes that come with Java begin with uppercase. You should follow this convention for classes you create as well.

Method and variables names begin with a lowercase letter followed by CamelCase. Class names begin with an uppercase letter followed by CamelCase.

TABLE 1.2 Default initialization values by type

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

Local variables can never have a scope larger than the method they are defined in. However, they can have a smaller scope. 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. Remember that blocks can contain other blocks. These smaller contained blocks can reference variables defined in the larger scoped blocks, but not vice versa.

All 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 (static) variables is even easier: they go into scope when declared like the other variables types. However, they stay in scope for the entire life of the program.

Local variables—in scope from declaration to end of block
Instance variables—in scope from declaration until object garbage collected
Class variables—in scope from declaration until program ends

TABLE 1.4 Elements of a class

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

Multiple classes can be defined in the same fi le, but only one of them is allowed to be public. The public class matches the name of the file. A file is also allowed to have neither class be public. As long as there isn’t more than one public class in a fi le, it is okay.

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program. You *do* need to know that System.gc() is not guaranteed to run, and you should be able to recognize when objects become eligible for garbage collection. Java provides a method called System.gc().It meekly *suggests* that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.

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

A reference may or may not be created on the heap. All references are the same size, no matter what their data type is, and are accessed by their variable name. Objects are always on the heap. They have no name and can only be accessed via a reference. Objects vary in size depending on their class definition.

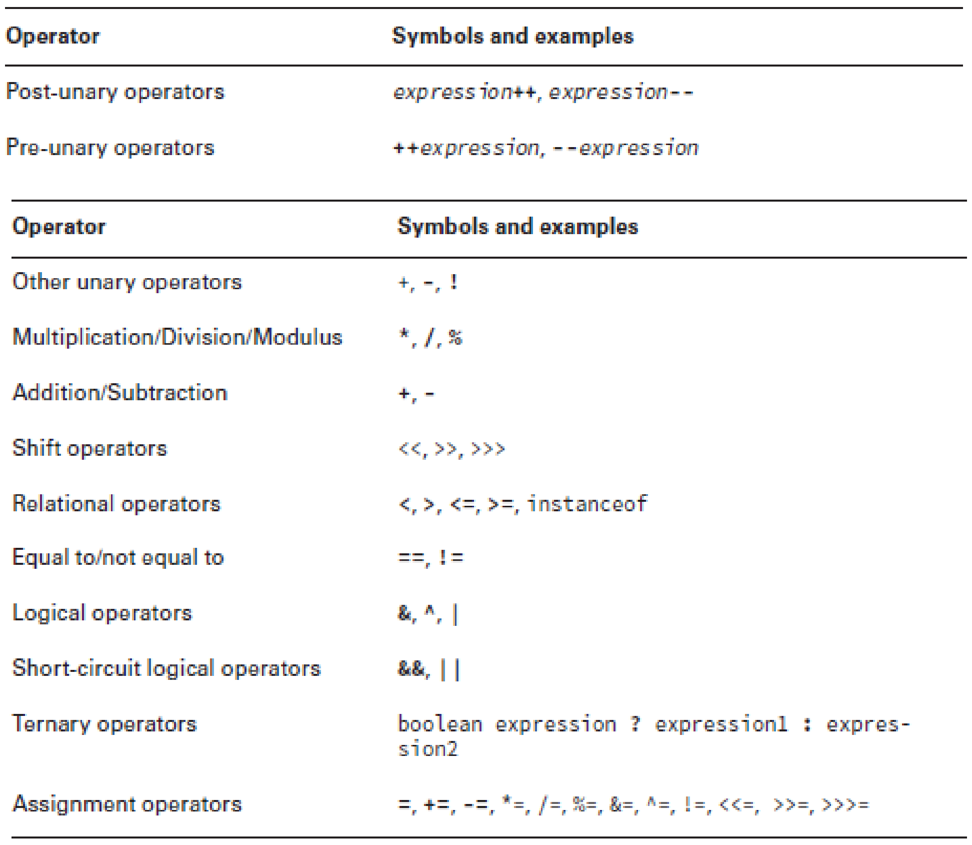
Java code runs inside the JVM. This creates a sandbox that makes it hard for Javacode to do evil things to the computer it is running on. So, Java is secure.

Operators and Statements

Remember that most of these structures require the evaluation of a particular boolean expression either for branching decisions or once per repetition. The switch statement is the only one that supports a variety of data types, including String variables as of Java 7.

With a for-each statement you don't need to explicitly write a boolean expression, since the compiler builds them implicitly. For clarity, we referred to an enhanced for loop as a for-each loop, but syntactically they are written as a for statement.

TABLE 2.1 Order of operator precedence



Operator	Symbols and examples
Post-unary operators	*expression*++, *expression*--
Pre-unary operators	++*expression*, --*expression*
Other unary operators	+, -, !
Multiplication/Division/Modulus	*, /, %
Addition/Subtraction	+, -
Shift operators	<<, >>, >>>
Relational operators	<, >, <=, >=, instanceof
Equal to/not equal to	==, !=
Logical operators	&, ^, |
Short-circuit logical operators	&&, ||
Ternary operators	boolean expression ? expression1 : expres-sion2
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, !=, <<=, >>=, >>>=

All of the arithmetic operators may be applied to any Java primitives, except Boolean and String. Furthermore, only the addition operators + and += may be applied to String values, which results in String concatenation.

Numeric Promotion Rules

- If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
- If one of the values is integral and the other is floating-point, Java will automatically promote the integral value to the floating-point value's data type.

- Smaller data types, namely byte, short, and char, are first promoted to int any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
- After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

For the third rule, note that unary operators are excluded from this rule. For example, applying ++ to a short value results in a short value.

What is the data type of x + y?
double x = 39.21;
float y = 2.1;
This is actually a trick question, as this code will not compile! Floating-point literals are assumed to be double, unless postfi xed with an f, as in 2.1f. If the value was set properly to 2.1f, then the promotion would be similar to the last example, with both operands being promoted to a double, and the result would be a double value.

What is the data type of x * y / z?
short x = 14;
float y = 13;
double z = 30;
In this case, we must apply all of the rules. First, x will automatically be promoted to int solely because it is a short and it is being used in an arithmetic binary operation. The promoted x value will then be automatically promoted to a float so that it can be multiplied with y. The result of x * y will then be automatically promoted to a double, so that it can be multiplied with z, resulting in a double value.

TABLE 2.2 Java unary operators

Unary operator	Description
+	Indicates a number is positive, although numbers are assumed to be positive in Java unless accompanied by a negative unary operator
-	Indicates a literal number is negative or negates an expression
++	Increments a value by 1
--	Decrements a value by 1
!	Inverts a Boolean's logical value

you cannot apply a negation operator, -, to a boolean expression, nor can you apply a logical complement operator, !, to a numeric expression.
int x = !5; // DOES NOT COMPILE
boolean y = -true; // DOES NOT COMPILE
boolean z = !0; // DOES NOT COMPILE

In Java 1 and true are not related in any way, just as 0 and false are not related.

Java will automatically promote from smaller to larger data types, as we saw in the previous section on arithmetic operators, but it will throw a compiler exception if it detects you are trying to convert from larger to smaller data types.

```
long t = 192301398193810323; // DOES NOT COMPILE
```
This statement does not compile because Java interprets the literal as an int and notices that the value is larger than int allows. The literal would need a postfi x L to be considered a long.

Casting primitives is required any time you are going from a larger numerical data type to a smaller numerical data type, or converting from a floating-point number to an integral value.
```
int x = (int)1.0;
short y = (short)1921222; // Stored as 20678
int z = (int)9L;
long t = 192301398193810323L;
```

Overflow is when a number is so large that it will no longer fit within the data type, so the system "wraps around" to the next lowest value and counts up from there. There's also an analogous *underflow*, when the number is too low to fi t in the data type.

```
short x = 10;
short y = 3;
short z = x * y; // DOES NOT COMPILE
```

```
short x = 10;
short y = 3;
short z = (short)(x * y); //Compiles without any issue
```

```
int x = 2, z = 3;
x = x * z; // Simple assignment operator
x *= z; // Compound assignment operator
```
The left-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable. Compound operators are useful for more than just shorthand—they can also save us from having to explicitly cast a value. For example, consider the following example, in which the last line will not compile due to the result being promoted to a long and assigned to an int variable:
```
long x = 10;
int y = 5;
y = y * x; // DOES NOT COMPILE
```
Based on the last two sections, you should be able to spot the problem in the last line. This last line could be fixed with an explicit cast to (int), but there's a better way using the compound assignment operator:
```
long x = 10;
```

```
int y = 5;
y *= x;
```
The compound operator will first cast x to a long, apply the multiplication of two long values, and then cast the result to an int. Unlike the previous example, in which the compiler threw an exception, in this example we see that the compiler will automatically cast the resulting value to the data type of the value on the left-hand side of the compound operator. One final thing to know about the assignment operator is that the result of the assignment is an expression in and of itself, equal to the value of the assignment. For example, the following snippet of code is perfectly valid, if not a little odd looking:
```
long x = 5;
long y = (x=3);
System.out.println(x); // Outputs 3
System.out.println(y); // Also, outputs 3
```
The key here is that (x=3) does two things. First, it sets the value of the variable x to be 3. Second, it returns a value of the assignment, which is also 3.

Four relational operators <, <=, >, >= are applied to numeric primitive data types only. If the two numeric operands are not of the same data type, the smaller one is promoted in the manner as previously discussed.

The fifth relational operator “instanceof” is applied to object references and classes or interfaces. “a instanceof b” returns True if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b.

The *logical operators*, (&), (|), and (^), may be applied to both numeric and boolean data types. When they’re applied to boolean data types, they’re referred to as *logical operators*. Alternatively, when they’re applied to numeric data types, they’re referred to as *bitwise operators*, as they perform bitwise comparisons of the bits that compose the number.

Finally, we present the conditional operators, && and ||, which are often referred to as short-circuit operators. The *short-circuit operators* are nearly identical to the logical operators, & and |, respectively, except that the right-hand side of the expression may never be evaluated if the fi nal result can be determined by the left-hand side of the expression.

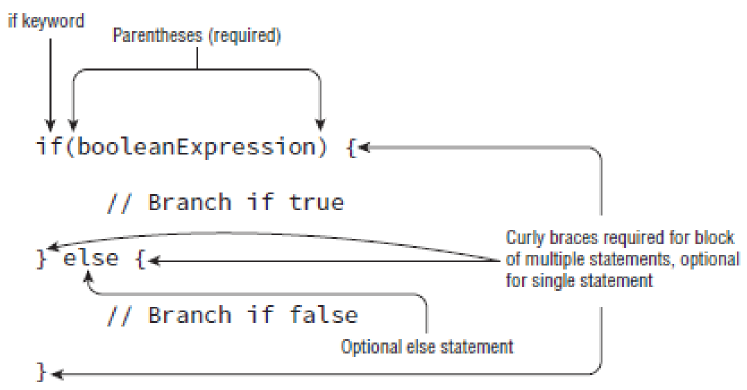
The equality operators are used in one of three scenarios:

- Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described.
- Comparing two boolean values.
- Comparing two objects, including null and String values.

```
boolean x = true == 3; // DOES NOT COMPILE
boolean y = false != "Giraffe"; // DOES NOT COMPILE
boolean z = 3 == "Kangaroo"; // DOES NOT COMPILE
```

For object comparison, the equality operator is applied to the references to the objects, not the objects they point to. Two references are equal if and only if they point to the same object, or both point to null.

FIGURE 2.3 The structure of an if-then-else statement



```
int x = 1;
if(x) { // DOES NOT COMPILE
...
}

int x = 1;
if(x = 5) { // DOES NOT COMPILE
...
}
```

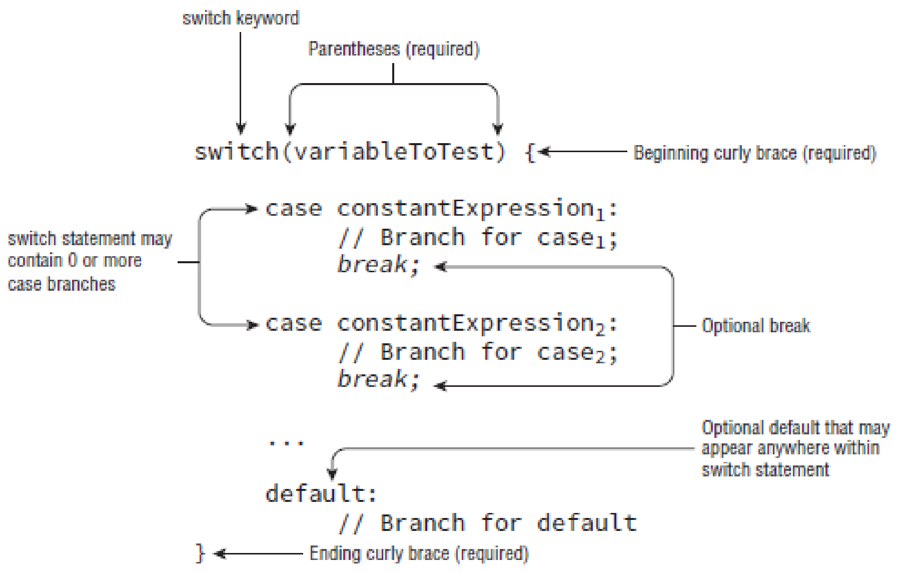
The conditional operator, ? :, otherwise known as the *ternary operator*, is the only operator that takes three operands and is of the form:
booleanExpression ? expression1 : expression2
The first operand must be a boolean expression, and the second and third can be any expression that returns a value. The ternary operation is really a condensed form of an ifthen-else statement that returns a value.

Note that it is often helpful for readability to add parentheses around the expressions in ternary operations, although it is certainly not required. There is no requirement that second and third expressions in ternary operations have the same data types, although it may come into play when combined with the assignment operator. Compare the following two statements:

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE
```

As of Java 7, only one of the right-hand expressions of the ternary operator will be evaluated at runtime, similar to the short-circuit operators.

FIGURE 2.4 The structure of a switch statement


switch keyword
Parentheses (required)
switch(variableToTest) {
Beginning curly brace (required)
switch statement may contain 0 or more case branches
case constantExpression1:
// Branch for case1;
break;
Optional break
case constantExpression2:
// Branch for case2;
break;
...
Optional default that may appear anywhere within switch statement
default:
// Branch for default
}
Ending curly brace (required)

Data types supported by switch statements include int and Integer, byte and Byte, short and Short, char and Character, String and enum values. Note that boolean and long, and their associated wrapper classes, are not supported by switch statements.

The values in each case statement must be compile-time constant values of the same data type as the switch value. This means you can use only literals, enum constants, or final constant variables of the same data type. By final constant, we mean that the variable must be marked with the final modifier and initialized with a literal value in the same expression in which it is declared.

There is no requirement that the case or default statements be in a particular order, unless you are going to have pathways that reach multiple sections of the switch block in a single execution. Even though the default block was before the case block, only the case block was executed. If you recall the defi nition of the default block, it is only branched to if there is no matching case value for the switch statement, regardless of its position within the switch statement.

```
private int getSortOrder(String firstName, final String lastName) {
String middleName = "Patricia";
final String suffix = "JR";
int id = 0;
switch(firstName) {
case "Test":
return 52;
case middleName: // DOES NOT COMPILE
id = 5;
break;
case suffix:
```

```
id = 0;
break;
case lastName: // DOES NOT COMPILE
id = 8;
break;
case 5: // DOES NOT COMPILE
id = 7;
break;
case 'J': // DOES NOT COMPILE
id = 10;
break;
case java.time.DayOfWeek.SUNDAY: // DOES NOT COMPILE
id=15;
break;
}
return id;
}
```

FIGURE 2.5 The structure of a while statement

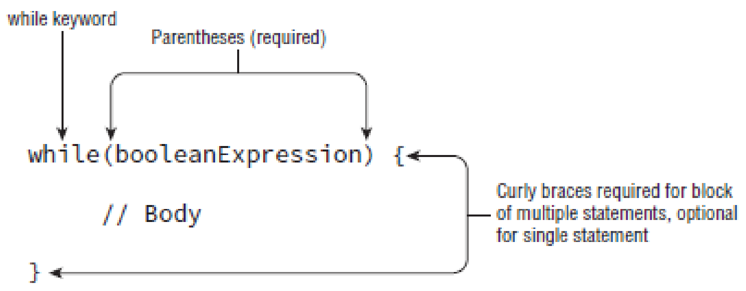

while keyword
Parentheses (required)
while(booleanExpression) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement

FIGURE 2.6 The structure of a do-while statement

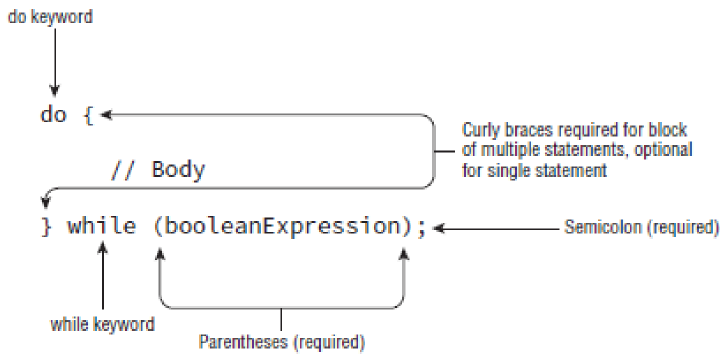
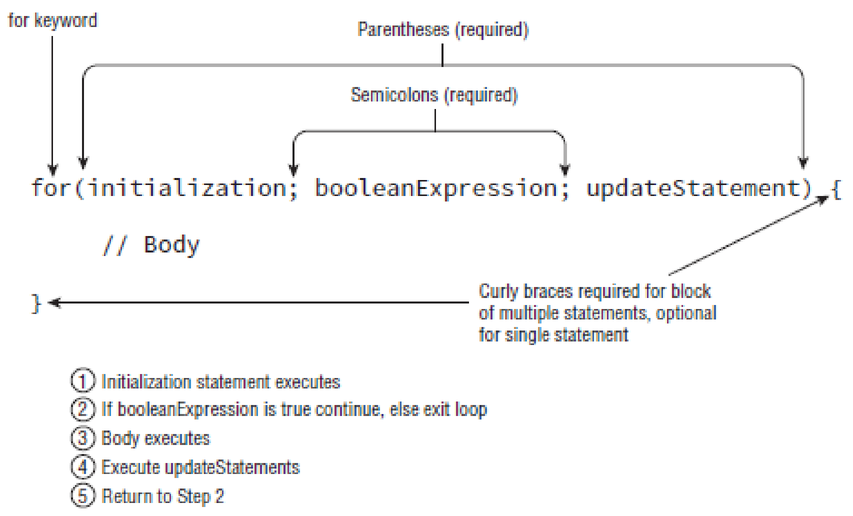

do keyword
do {
// Body
} while (booleanExpression);
Curly braces required for block of multiple statements, optional for single statement
Semicolon (required)
while keyword
Parentheses (required)

FIGURE 2.7 The structure of a basic for statement


for keyword
Parentheses (required)
Semicolons (required)
for(initialization; booleanExpression; updateStatement) {
// Body
}
Curly braces required for block of multiple statements, optional for single statement
1 Initialization statement executes
2 If booleanExpression is true continue, else exit loop
3 Body executes
4 Execute updateStatements
5 Return to Step 2

The initialization and update sections may contain multiple statements, separated by commas. Variables declared in the initialization block of a for loop have limited scope and are only accessible within the for loop.

Creating an Infinite Loop

```
for( ; ; ) {
System.out.println("Hello World");
}
```

Although this for loop may look like it will throw a compiler error, it will in fact compile and run without issue. It is actually an infinite loop that will print the same statement repeatedly.

Adding Multiple Terms to the for Statement

```
int x = 0;
for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {
System.out.print(y + " ");
}
System.out.print(x);
```

Redeclaring a Variable in the Initialization Block

```
int x = 0;
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // DOES NOT COMPILE
System.out.print(x + " ");
}
```

Using Incompatible Data Types in the Initialization Block

```
for(long y = 0, int x = 4; x < 5 && y<10; x++, y++) { // DOES NOT COMPILE
System.out.print(x + " ");
}
```

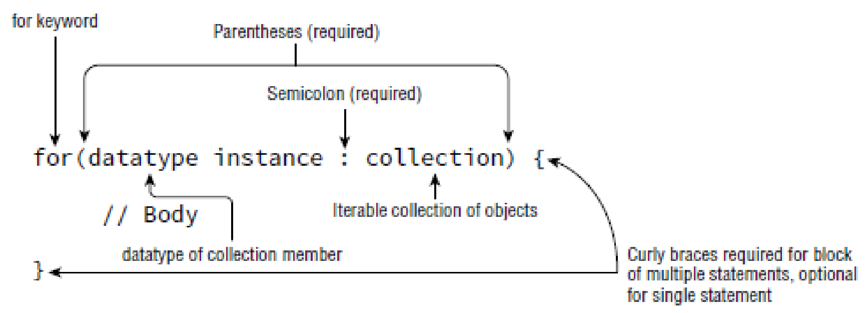
The variables in the initialization block must all be of the same type.

Using Loop Variables Outside the Loop

```
for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {
System.out.print(y + " ");
}
System.out.print(x); // DOES NOT COMPILE
```

Enhanced for loop is designed for iterating over arrays and Collection objects. Enhanced for loop is also called as a *for-each* loop

FIGURE 2.8 The structure of an enhancement for statement


for keyword
Parentheses (required)
Semicolon (required)
for(datatype instance : collection) {
// Body
}
Iterable collection of objects
datatype of collection member
Curly braces required for block of multiple statements, optional for single statement

The for-each loop declaration is composed of an initialization section and an object to be iterated over. The right-hand side of the for-each loop statement must be a built-in Java array or an object whose class implements java.lang.Iterable, which includes most of the Java Collections framework. The left-hand side of the for-each loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement. On each iteration of the loop, the named variable on the left-hand side of the statement is assigned a new value from the array or collection on the right-hand side of the statement.

```
final String[] names = new String[3];
names[0] = "Lisa";
names[1] = "Kevin";
names[2] = "Roger";
for(String name : names) {
System.out.print(name + ", ");
}
```

This code will compile and print:

```
Lisa, Kevin, Roger,
```

```
java.util.List<String> values = new java.util.ArrayList<String>();
values.add("Lisa");
values.add("Kevin");
values.add("Roger");
```

```
for(String value : values) {
System.out.print(value + ", ");
}
```

This code will compile and print the same values:

```
Lisa, Kevin, Roger,
```

When you see a for-each loop on the exam, make sure the right-hand side is an array or Iterable object and the left-hand side has a matching type.

```
String names = "Lisa";
for(String name : names) { // DOES NOT COMPILE
System.out.print(name + " ");
}
```

In this example, the String names is not an array, nor does it implement java.lang.Iterable, so the compiler will throw an exception since it does not know how to iterate over the String.

```
String[] names = new String[3];
for(int name : names) { // DOES NOT COMPILE
System.out.print(name + " ");
}
```

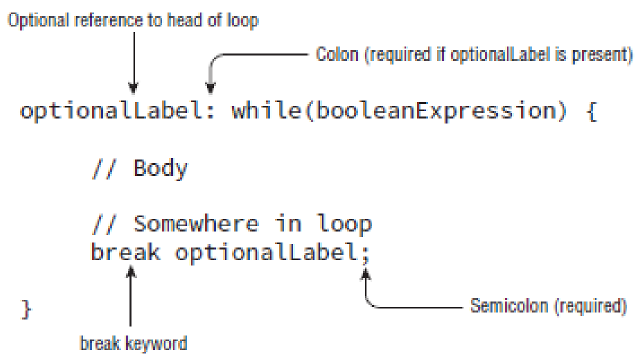
This code will fail to compile because the left-hand side of the for-each statement does not defi ne an instance of String. Notice that in this last example, the array is initialized with three null pointer values. In and of itself, that will not cause the code to not compile, as a corrected loop would just output null three times.

While the for-each statement is convenient for working with lists in many cases, it does hide access to the loop iterator variable. If we wanted to print only the comma between names, we could convert the example into a standard for loop. It is also common to use a standard for loop over a for-each loop if comparing multiple elements in a loop within a single iteration.

```
int[][] myComplexArray = {{5,2,1,3},{3,9,8,9},{5,7,12,7}};
for(int[] mySimpleArray : myComplexArray) {
for(int i=0; i<mySimpleArray.length; i++) {
System.out.print(mySimpleArray[i]+"\t");
}
System.out.println();
}
```

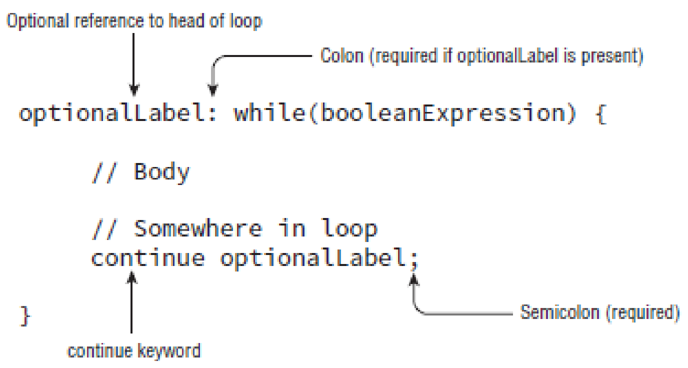
A *label* is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded by a colon (:). When dealing with only one loop, they add no value, but they are extremely useful in nested environments. Optional labels are often only used in loop structures. For formatting, labels follow the same rules for identifiers. For readability, they are commonly expressed in uppercase, with underscores between words, to distinguish them from regular variables.

FIGURE 2.9 The structure of a break statement


Optional reference to head of loop
Colon (required if optionalLabel is present)
optionalLabel: while(booleanExpression) {
// Body
// Somewhere in loop
break optionalLabel;
}
Semicolon (required)
break keyword

The break statement can take an optional label parameter. Without a label parameter, the break statement will terminate the nearest inner loop it is currently in the process of executing. The optional label parameter allows us to break out of a higher level outer loop.

FIGURE 2.10 The structure of a continue statement


Optional reference to head of loop
Colon (required if optionalLabel is present)
optionalLabel: while(booleanExpression) {
// Body
// Somewhere in loop
continue optionalLabel;
}
Semicolon (required)
continue keyword

The break statement transfers control to the enclosing statement, the continue statement transfers control to the boolean expression that determines if the loop should continue. In other words, it ends the current iteration of the loop. Also like the break statement, the continue statement is applied to the nearest inner loop under execution using optional label statements to override this behavior.

TABLE 2.5 Advanced flow control usage

	Allows optional labels	Allows *break* statement	Allows *continue* statement
`if`	Yes *	No	No
`while`	Yes	Yes	Yes
`do while`	Yes	Yes	Yes
`for`	Yes	Yes	Yes
`switch`	Yes	Yes	No

* Labels are allowed for any block statement, including those that are preceded with an `if-then` statement.

Core Java APIs

Strings are immutable sequences of characters. The new operator is optional. The concatenation operator (+) creates a new String with the content of the first String followed by the content of the second String. If either operand involved in the + expression is a String, concatenation is used; otherwise, addition is used. String literals are stored in the string pool. The String class has many methods. You need to know charAt(), concat(), endsWith(), equals(), equalsIgnoreCase(), indexOf(), length(), replace(), startsWith(), substring(), toLowerCase(), toUpperCase(), and trim().

StringBuilders are mutable sequences of characters. Most of the methods return a reference to the current object to allow method chaining. The StringBuilder class has many methods. You need to know append(), charAt(), delete(), deleteCharAt(), indexOf(), insert(), length(), reverse(), substring(), and toString(). StringBuffer is the same as StringBuilder except that it is thread safe.

Calling == on String objects will check whether they point to the same object in the pool. Calling == on StringBuilder references will check whether they are pointing to the same StringBuilder object. Calling equals() on String objects will check whether the sequence of characters is the same. Calling equals() on StringBuilder objects will check whether they are pointing to the same object rather than looking at the values inside.

An array is a fixed-size area of memory on the heap that has space for primitives or pointers to objects. You specify the size when creating it—for example, int[] a = new int[6];. Indexes begin with 0 and elements are referred to using a[0]. The Arrays.sort() method sorts an array. Arrays.binarySearch() searches a sorted array and returns the index of a match. If no match is found, it negates the position where the element would need to be inserted and subtracts 1. Methods that are passed varargs (…) can be used as if a normal array was passed in. In a multidimensional array, the second-level arrays and beyond can be different sizes.

An ArrayList can change size over its life. It can be stored in an ArrayList or List reference. Generics can specify the type that goes in the ArrayList. You need to know the methods add(), clear(), contains(), equals(), isEmpty(), remove(), set(), and size(). Although an ArrayList is not allowed to contain primitives, Java will autobox parameters passed in to the proper wrapper type. Collections.sort() sorts an ArrayList.

A LocalDate contains just a date, a LocalTime contains just a time, and a LocalDateTime contains both a date and time. All three have private constructors and are created using LocalDate.now() or LocalDate.of() (or the equivalents for that class). Dates and times can be manipulated using plusXXX or minusXXX methods. The Period class represents a number of days, months, or years to add or subtract from a LocalDate or LocalDateTime. DateTimeFormatter is used to output dates and times in the desired format. The date and time classes are all immutable, which means the return value must be used.

Strings are immutable. Pay special attention to the fact that indexes are zero based and that substring() gets the string up until right before the index of the second parameter.

StringBuilder is mutable. Know that substring() does not change the value of a StringBuilder whereas append(), delete(), and insert() do change it. Also note that most StringBuilder methods return a reference to the current instance of StringBuilder.

== checks object equality. equals() depends on the implementation of the object it is being called on. For Strings, equals() checks the characters inside of it.

In Java, these two snippets both create a String:
```
String name = "Fluffy";
String name = new String("Fluffy");
```
Both give you a reference variable of type name pointing to the String object "Fluffy". They are subtly different, as you'll see in the section "String Pool". For now, just remember that the String class is special and doesn't need to be instantiated with new.

1 + 2 is clearly 3. But what is "1" + "2"? It's actually "12" because Java combines the two String objects. Placing one String before the other String and combining them together is called string *concatenation*.

- If both operands are numeric, + means numeric addition.
- If either operand is a String, + means concatenation. Other operand will be converted to String, irrespective of its actual data type.
- The expression is evaluated left to right.

```
System.out.println(1 + 2 + "c"); // 3c
```

Once a String object is created, it is not allowed to change. It cannot be made larger or smaller, and you cannot change one of the characters inside it. *Mutable* is another word for changeable. *Immutable* is the opposite—an object that can't be changed once it's created.

```
String s1 = "1";
String s2 = s1.concat("2");
s2.concat("3");
System.out.println(s2);
```
Did you say "12"? Good. The trick is to see if you forget that the String class is immutable by throwing a method at you.

The *string pool*, also known as the intern pool, is a location in the Java virtual machine (JVM) that collects all these strings. The string pool contains literal values that appear in your program. For example, "name" is a literal and therefore goes into the string pool. myObject.toString() is a string but not a literal, so it does not go into the string pool. Strings not in the string pool are garbage collected just like any other object.
Remember back when we said these two lines are subtly different?

String name = "Fluffy";
String name = new String("Fluffy");
The former says to use the string pool normally. The second says “No, JVM. I really don’t want you to use the string pool. Please create a new object for me even though it is less efficient”

Important String Methods
length()
The method length() returns the number of characters in the String. The method signature is as follows:
int length()

charAt()
The method charAt() lets you query the string to fi nd out what character is at a specific index. The method signature is as follows:
char charAt(int index)
The following code shows how to use charAt():
String string = "animals";
System.out.println(string.charAt(7)); // throws exception java.lang.StringIndexOutOfBoundsException: String index out of range: 7

indexOf()
The method indexOf()looks at the characters in the string and finds the first index that matches the desired value. indexOf can work with an individual character or a whole String as input. It can also start from a requested position. The method signatures are as follows:
int indexOf(char ch)
int indexOf(char ch, index fromIndex)
int indexOf(String str)
int indexOf(String str, index fromIndex)
The following code shows how to use indexOf():
String string = "animals";
System.out.println(string.indexOf('a')); // 0
System.out.println(string.indexOf("al")); // 4
System.out.println(string.indexOf('a', 4)); // 4
System.out.println(string.indexOf("al", 5)); // -1
Unlike charAt(), the indexOf() method doesn’t throw an exception if it can’t fi nd a match. indexOf() returns –1 when no match is found.

substring()
The method substring() also looks for characters in a string. It returns parts of the string. The first parameter is the index to start with for the returned string. As usual, this is a zero-based index. There is an optional second parameter, which is the end index you want to stop at. Notice we said “stop at” rather than “include.” This means the endIndex parameter is allowed to be 1 past the end of the sequence if you want to stop at the end of the sequence. That would be redundant, though, since you could omit the second parameter entirely in that case. In your own code, you

want to avoid this redundancy. Don't be surprised if the exam uses it though. The method signatures are as follows:

```
int substring(int beginIndex)
int substring(int beginIndex, int endIndex)
```

The following code shows how to use substring():

```
String string = "animals";
System.out.println(string.substring(3)); // mals
System.out.println(string.substring(string.indexOf('m'))); // mals
System.out.println(string.substring(3, 4)); // m
System.out.println(string.substring(3, 7)); // mals
System.out.println(string.substring(3, 3)); // empty string
System.out.println(string.substring(3, 2)); // throws exception
System.out.println(string.substring(3, 8)); // throws exception
```

toLowerCase()* and *toUpperCase()

```
String toLowerCase(String str)
String toUpperCase(String str)
```

The following code shows how to use these methods:

```
String string = "animals";
System.out.println(string.toUpperCase()); // ANIMALS
System.out.println("Abc123".toLowerCase()); // abc123
```

These methods leave alone any characters other than letters.Also, remember that strings are immutable, so the original string stays the same.

equals()* and *equalsIgnoreCase()

The equals() method checks whether two String objects contain exactly the same characters in the same order. The equalsIgnoreCase() method checks whether two String objects contain the same characters with the exception that it will convert the characters' case if needed. The method signatures are as follows:

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
```

The following code shows how to use these methods:

```
System.out.println("abc".equals("ABC")); // false
System.out.println("ABC".equals("ABC")); // true
System.out.println("abc".equalsIgnoreCase("ABC")); // true
```

startsWith()* and *endsWith()

```
boolean startsWith(String prefix)
boolean endsWith(String suffix)
```

The following code shows how to use these methods:

```
System.out.println("abc".startsWith("a")); // true
System.out.println("abc".startsWith("A")); // false
System.out.println("abc".endsWith("c")); // true
System.out.println("abc".endsWith("a")); // false
```

contains()
The contains() method also looks for matches in the String. It isn't as particular as startsWith() and endsWith(). the match can be anywhere in the String. The method signature is as follows:
boolean contains(String str)
The following code shows how to use these methods:
System.out.println("abc".contains("b")); // true
System.out.println("abc".contains("B")); // false
Again, we have a case-sensitive search in the String. The contains() method is a convenience method so you don't have to write **str.indexOf(otherString) != -1**.

replace()
The replace() method does a simple search and replace on the string. There's a version that takes char parameters as well as a version that takes CharSequence parameters. A CharSequence is a general way of representing several classes, including String and StringBuilder. It's called an interface, which we'll cover in Chapter 5, "Class Design."
The method signatures are as follows:
String replace(char oldChar, char newChar)
String replace(CharSequence oldChar, CharSequence newChar)
The following code shows how to use these methods:
System.out.println("abcabc".replace('a', 'A')); // AbcAbc
System.out.println("abcabc".replace("a", "A")); // AbcAbc
The first example uses the first method signature, passing in char parameters. The second example uses the second method signature, passing in String parameters.

trim()
The trim() method removes whitespace from the beginning and end of a String. In terms of the exam, whitespace consists of spaces along with the \t (tab) and \n (newline) characters. Other characters, such as \r (carriage return), are also included in what gets trimmed. The method signature is as follows:
public String trim()
The following code shows how to use this method:
System.out.println("abc".trim()); // abc
System.out.println("\t a b c\n".trim()); // a b c

You'll see code using a technique called method chaining. Here's an example:
String result = "AniMaL ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
It creates four String objects and outputs Animal. To read code that uses method chaining, start at the left and evaluate the first method. Then call the next method on the returned value of the first method. Keep going until you get to the semicolon. Remember that String is immutable.

StringBuilder is not immutable.
15: StringBuilder alpha = new StringBuilder();

```
16: for(char current = 'a'; current <= 'z'; current++)
17: alpha.append(current);
18: System.out.println(alpha);
```
On line 15, a new StringBuilder object is instantiated. The call to append() on line 17 adds a character to the StringBuilder object each time through the for loop and appends the value of current to the end of alpha. This code reuses the same StringBuilder without creating an interim String each time.

When we chained String method calls, the result was a new String with the answer. Chaining StringBuilder objects doesn’t work this way. Instead, the StringBuilder changes its own state and returns a reference to itself!
Let’s look at an example to make this clearer:
```
4: StringBuilder sb = new StringBuilder("start");
5: sb.append("+middle"); // sb = "start+middle"
6: StringBuilder same = sb.append("+end"); // "start+middle+end"
```
Line 5 adds text to the end of sb. It also returns a reference to sb, which is ignored. Line 6 also adds text to the end of sb and returns a reference to sb. This time the reference is stored in same—which means sb and same point to the exact same object and would print out the same value.

```
4: StringBuilder a = new StringBuilder("abc");
5: StringBuilder b = a.append("de");
6: b = b.append("f").append("g");
7: System.out.println("a=" + a);
8: System.out.println("b=" + b);
```
Did you say both print "abcdefg"? Good. There’s only one StringBuilder object here. We know that because new StringBuilder() was called only once.

There are three ways to construct a StringBuilder:
```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```
The fi rst says to create a StringBuilder containing an empty sequence of characters and assign sb1 to point to it. The second says to create a StringBuilder containing a specific value and assign sb2 to point to it. For the fi rst two, it tells Java to manage the implementation details. The fi nal example tells Java that we have some idea of how big the eventual value will be and would like the StringBuilder to reserve a certain number of slots for characters.

Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold. Since a String is immutable, the size and capacity are the same. The number of characters appearing in the String is both the size and capacity. For StringBuilder, Java knows the size is likely to change as the object is used. When StringBuilder is constructed, it may start at the default capacity (which happens to be 16) or one of the programmer’s choosing.

Important *StringBuilder* Methods

charAt(), indexOf(), length(),* and *substring()

These four methods work exactly the same as in the String class. substring() returns a String rather than a StringBuilder.

append()

The append() method is by far the most frequently used method in StringBuilder. In fact, it is so frequently used that we just started using it without comment. Luckily, this method does just what it sounds like: it adds the parameter to the StringBuilder and returns a reference to the current StringBuilder. One of the method signatures is as follows:

```
StringBuilder append(String str)
```

Notice that we said *one* of the method signatures. There are more than 10 method signatures that look similar but that take different data types as parameters. All those methods are provided so you can write code like this:

```
StringBuilder sb = new StringBuilder().append(1).append('c');
sb.append("-").append(true);
System.out.println(sb); // 1c-true
```

Nice method chaining, isn't it? append() is called directly after the constructor. By having all these method signatures, you can just call append() without having to convert your parameter to a String first.

insert()

The insert() method adds characters to the StringBuilder at the requested index and returns a reference to the current StringBuilder. Just like append(), there are lots of method signatures for different types. Here's one:

```
StringBuilder insert(int offset, String str)
```

Pay attention to the offset in these examples. It is the index where we want to insert the requested parameter.

```
3: StringBuilder sb = new StringBuilder("animals");
4: sb.insert(7, "-"); // sb = animals-
5: sb.insert(0, "-"); // sb = -animals-
6: sb.insert(4, "-"); // sb = -ani-mals
7: System.out.println(sb);
```

If we give invalid index like -1 or 8 before 4th line we will get the exception "Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range"

delete()* and *deleteCharAt()

The delete() method is the opposite of the insert() method. It removes characters from the sequence and returns a reference to the current StringBuilder. The deleteCharAt() method is convenient when you want to delete only one character. The method signatures are as follows:

```
StringBuilder delete(int start, int end)
StringBuilder deleteCharAt(int index)
```

The following code shows how to use these methods:

```
StringBuilder sb = new StringBuilder("abcdef");
sb.delete(1, 3); // sb = adef
sb.deleteCharAt(5); // throws an exception
```
First, we delete the characters starting with index 1 and ending right before index 3. This gives us adef. Next, we ask Java to delete the character at position 5. However, the remaining value is only four characters long, so it throws a StringIndexOutOfBoundsException.

reverse()

After all that, it's time for a nice, easy method. The reverse() method does just what it sounds like: it reverses the characters in the sequences and returns a reference to the current StringBuilder. The method signature is as follows:
```
StringBuilder reverse()
```
The following code shows how to use this method:
```
StringBuilder sb = new StringBuilder("ABC");
sb.reverse();
System.out.println(sb); //prints CBA
```

toString()

The last method converts a StringBuilder into a String. The method signature is as follows:
```
String toString()
```
The following code shows how to use this method:
```
String s = sb.toString();
```
Often StringBuilder is used internally for performance purposes but the end result needs to be a String. For example, maybe it needs to be passed to another method that is expecting a String.

StringBuilder* vs. *StringBuffer

When writing new code that concatenates a lot of String objects together, you should use StringBuilder. StringBuilder was added to Java in Java 5. If you come across older code, you will see StringBuffer used for this purpose. StringBuffer does the same thing but more slowly because it is thread safe.

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two); // false
System.out.println(one == three); // true
```
Since this example isn't dealing with primitives, we know to look for whether the references are referring to the same object.

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y); // true
```

Remember that Strings are immutable and literals are pooled. The JVM created onlyone literal in memory. x and y both point to the same location in memory; therefore, the statement outputs true.

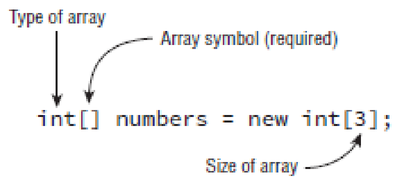
String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x == z); // false
In this example, we don't have two of the same String literal. Although x and z happen to evaluate to the same string, one is computed at runtime. Since it isn't the same at compile-time, a new String object is created. You can even force the issue by creating a new String:
String x = new String("Hello World");
String y = "Hello World";
System.out.println(x == y); // false
Since you have specifically requested a different String object, the pooled value isn't shared.

String x = "Hello World";
String z = " Hello World".trim();
System.out.println(x.equals(z)); // true
If a class doesn't have an equals method, Java determines whether the references point to the same object which is exactly what == does. In case you are wondering, the authors of StringBuilder did not implement equals(). If you call equals() on two StringBuilder instances, it will check reference equality.

A String is implemented as an array with some methods that you might want to use when dealing with characters specifi cally. A StringBuilder is implemented as an array where the array object is replaced with a new bigger array object when it runs out of space to store all the characters. A big difference is that an array can be of any other Java type. If we didn't want to use a String for some reason, we could use an array of char primitives directly.

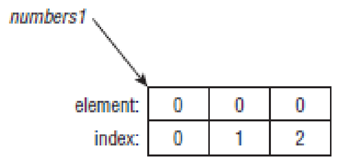
An array is an ordered list. It can contain duplicates.

FIGURE 3.2 The basic structure of an array



When using this form to instantiate an array, set all the elements to the default value for that type.

FIGURE 3.3 An empty array



Another way to create an array is to specify all the elements it should start out with:
int[] numbers2 = new int[] {42, 55, 99};
In this example, we also create an int array of size 3. This time, we specify the initial values of those three elements instead of using the defaults.

Java recognizes that this expression is redundant. Since you are specifying the type of the array on the left side of the equal sign, Java already knows the type. And since you are specifying the initial values, it already knows the size. As a shortcut, Java lets you write this:
int[] numbers2 = {42, 55, 99};
This approach is called an anonymous array. It is anonymous because you don't specify the type and size. Finally, you can type the [] before or after the name, and adding a space is optional. This means that all four of these statements do the exact same thing:
int[] numAnimals;
int [] numAnimals2;
int numAnimals3[];
int numAnimals4 [];

int[] ids, types;
The correct answer is two variables of type int[].
int ids[], types;
All we did was move the brackets, but it changed the behavior. This time we get one variable of type int[] and one variable of type int.

```
public class ArrayType {
public static void main(String args[]) {
String [] bugs = { "cricket", "beetle", "ladybug" };
String [] alias = bugs;
System.out.println(bugs.equals(alias)); // true
System.out.println(bugs.toString()); // [Ljava.lang.String;@160bc7c0
} }
```
The second print statement is even more interesting. What on earth is [Ljava.lang.String;@160bc7c0? You don't have to know this for the exam, but [L means it is an array, java.lang.String is the reference type, and 160bc7c0 is the hash code. Since Java 5, Java has provided a method that prints an array nicely: java.util.Arrays.toString(bugs) would print [cricket, beetle, ladybug].

```
3: String[] strings = { "stringValue" };
```

```
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder(); // DOES NOT COMPILE
7: objects[0] = new StringBuilder(); // careful!
```

Line 3 creates an array of type String. Line 4 doesn't require a cast because Object is a broader type than String. On line 5, a cast is needed because we are moving to a more specific type. Line 6 doesn't compile because a String[] only allows String objects and StringBuilder is not a String. Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A StringBuilder object can clearly go in an Object[]. The problem is that we don't actually have an Object[]. We have a String[] referred to from an Object[] variable. At runtime, the code throws an ArrayStoreException.

```
4: String[] mammals = {"monkey", "chimp", "donkey"};
5: System.out.println(mammals.length); // 3
```

Line 5 tells us how many elements the array can hold. length does not consider what is in the array; it only considers how many slots have been allocated. Note that length is a variable, but not a method. So, we need to avoid using parenthesis here.

Arrays is the first class provided by Java we have used that requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.* // import whole package including Arrays
import java.util.Arrays; // import just Arrays
```

There is one exception, although it doesn't come up often on the exam. You can write java.util.Arrays every time it is used in the class instead of specifying it as an import.

```
int[] numbers = { 6, 9, 1 };
Arrays.sort(numbers);
for (int i = 0; i < numbers.length; i++)
 System.out.print (numbers[i] + " ");
```

The result is 1 6 9, as you should expect it to be. Notice that we had to loop through the output to print the values in the array. Just printing the array variable directly would give the annoying hash of [I@2bd9c3e7.

Try this again with String types:

```
String[] strings = { "10", "9", "100" };
Arrays.sort(strings);
for (String string : strings)
System.out.print(string + " ");
```

This time the result might not be what you expect. This code outputs 10 100 9. The problem is that String sorts in alphabetic order, and 1 sorts before 9. (Numbers sort before letters and uppercase sorts before lowercase, in case you were wondering.)

Java also provides a convenient way to search—but only if the array is already sorted.

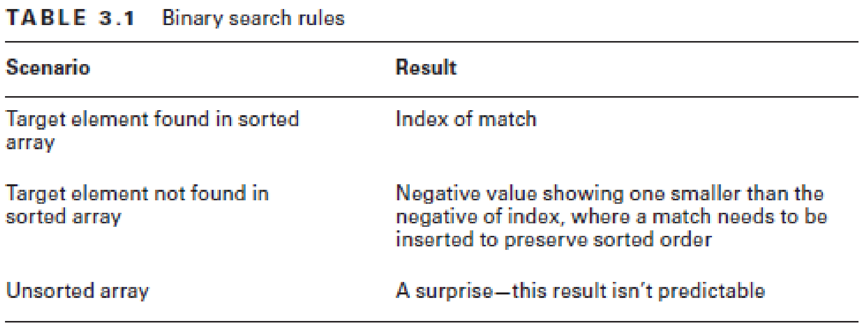


TABLE 3.1 Binary search rules

Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise—this result isn't predictable

```
3: int[] numbers = {2,4,6,8};
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

```
public static void main(String... args) // varargs
```

This example uses a syntax called varargs (variable arguments). You can use a variable defined using varargs as if it were a normal array. For example args.length and args[0] are legal.

```
int[][] vars1; // 2D array
int vars2 [][]; // 2D array
int[] vars3[]; // 2D array
int[] vars4 [], space [][]; // a 2D AND a 3D array
```

```
String [][] rectangle = new String[3][2];
```

While that array happens to be rectangular in shape, an array doesn't need to be. Consider this one:

```
int[][] differentSize = {{1, 4}, {3}, {9,8,7}};
```

We still start with an array of three elements. However, this time the elements in the next level are all different sizes. One is of length 2, the next length 1, and the last length 3.

Another way to create an asymmetric array is to initialize just an array's first dimension, and define the size of each array component in a separate statement:

```
int [][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

```
for (int[] inner : twoD) {
for (int num : inner)
System.out.print(num + " ");
System.out.println();
}
```

An array has one glaring shortcoming: you have to know how many elements will be in the array when you create it and then you are stuck with that choice. Just like a StringBuilder, ArrayList can change size at runtime as needed. Like an array, an ArrayList is an ordered sequence that allows duplicates. As when we used Arrays.sort, ArrayList requires an import. To use it, you must have either of the following two statements in your class:

```
import java.util.* // import whole package including ArrayList
import java.util.ArrayList; // import just ArrayList
```

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

The first says to create an ArrayList containing space for the default number of elements but not to fi ll any slots yet. The second says to create an ArrayList containing a specific number of slots, but again not to assign any. The final example tells Java that we want to make a copy of another ArrayList. We copy both the size and contents of that ArrayList. Granted, list2 is empty in this example so it isn't particularly interesting. These examples were the old pre–Java 5 way of creating an ArrayList.

Java 5 introduced *generics*, which allow you to specify the type of class that the ArrayList will contain.

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Java 5 allows you to tell the compiler what the type would be by specifying it between < and >. Starting in Java 7, you can even omit that type from the right side. The < and > are still required, though. This is called the diamond operator because <> looks like a diamond. Just when you thought you knew everything about creating an ArrayList, there is one more thing you need to know. ArrayList implements an interface called List. In other words, an ArrayList is a List. Just know that you can store an ArrayList in a List reference variable but not vice versa. The reason is that List is an interface and interfaces can't be instantiated.

```
List<String> list6 = new ArrayList<>();
ArrayList<String> list7 = new List<>(); // DOES NOT COMPILE
```

Before reading any further, you are going to see something new in the method signatures: a "class" named E. Don't worry—it isn't really a class. E is used by convention in generics to mean "any class that this array can hold." If you didn't specify a type when creating the ArrayList, E means Object. Otherwise, it means the class you put between < and >. You should also know that ArrayList implements toString() so you can easily see the contents just by printing it. Arrays do not do produce such pretty output.

add()

```
boolean add(E element)
```

void add(int index, E element)
Don't worry about the boolean return value. It always returns true. It is there because other classes in the collections family need a return value in the signature when adding an element.

```
ArrayList list = new ArrayList();
list.add("hawk"); // [hawk]
list.add(Boolean.TRUE); // [hawk, true]
System.out.println(list); // [hawk, true]
```

add() does exactly what we expect: it stores the String in the no longer empty ArrayList. It then does the same thing for the boolean. This is okay because we didn't specify a type for ArrayList; therefore, the type is Object, which includes everything except primitives. Now, let's use generics to tell the compiler we only want to allow String objects in our ArrayList:

```
ArrayList<String> safer = new ArrayList<>();
safer.add("sparrow");
safer.add(Boolean.TRUE); // DOES NOT COMPILE
```

```
4: List<String> birds = new ArrayList<>();
5: birds.add("hawk"); // [hawk]
6: birds.add(1, "robin"); // [hawk, robin]
7: birds.add(0, "blue jay"); // [blue jay, hawk, robin]
8: birds.add(1, "cardinal"); // [blue jay, cardinal, hawk, robin]
9: System.out.println(birds); // [blue jay, cardinal, hawk, robin]
```

remove()
The remove() methods remove the first matching value in the ArrayList or remove the element at a specified index. The method signatures are as follows:
boolean remove(Object object)
E remove(int index)
This time the boolean return value tells us whether a match was removed. The E return type is the element that actually got removed. The following shows how to use these methods:

```
3: List<String> birds = new ArrayList<>();
4: birds.add("hawk"); // [hawk]
5: birds.add("hawk"); // [hawk, hawk]
6: System.out.println(birds.remove("cardinal")); // prints false
7: System.out.println(birds.remove("hawk")); // prints true
8: System.out.println(birds.remove(0)); // prints hawk
9: System.out.println(birds); // []
```

set()
The set() method changes one of the elements of the ArrayList without changing the size. The method signature is as follows:
E set(int index, E newElement)
The E return type is the element that got replaced. The following shows how to use this method:

```
15: List<String> birds = new ArrayList<>();
16: birds.add("hawk"); // [hawk]
```

```
System.out.println(birds.size()); // 1
birds.set(0, "robin"); // [robin]
System.out.println(birds.size()); // 1
birds.set(1, "robin"); // IndexOutOfBoundsException
```

isEmpty() and size()

The isEmpty() and size() methods look at how many of the slots are in use. The method signatures are as follows:

```
boolean isEmpty()
int size()
```

clear()

The clear() method provides an easy way to discard all elements of the ArrayList. The method signature is as follows:

```
void clear()
```

contains()

The contains() method checks whether a certain value is in the ArrayList. The method signature is as follows:

```
boolean contains(Object object)
```

The following shows how to use this method:

```
List<String> birds = new ArrayList<>();
birds.add("hawk"); // [hawk]
System.out.println(birds.contains("hawk")); // true
System.out.println(birds.contains("robin")); // false
```

This method calls equals() on each element of the ArrayList to see whether there are any matches. Since String implements equals(), this works out well.

equals()

Finally, ArrayList has a custom implementation of equals() so you can compare two lists to see if they contain the same elements in the same order.

```
boolean equals(Object object)
```

The following shows an example:

```
List<String> one = new ArrayList<>();
List<String> two = new ArrayList<>();
System.out.println(one.equals(two)); // true
one.add("a"); // [a]
System.out.println(one.equals(two)); // false
two.add("a"); // [a]
System.out.println(one.equals(two)); // true
one.add("b"); // [a,b]
two.add(0, "b"); // [b,a]
System.out.println(one.equals(two)); // false
```

Up to now, we’ve only put String objects in the ArrayList. What happens if we want to put primitives in? Each primitive type has a wrapper class, which is an object type that corresponds to the primitive. Table 3.2 lists all the wrapper classes along with the constructor for each.

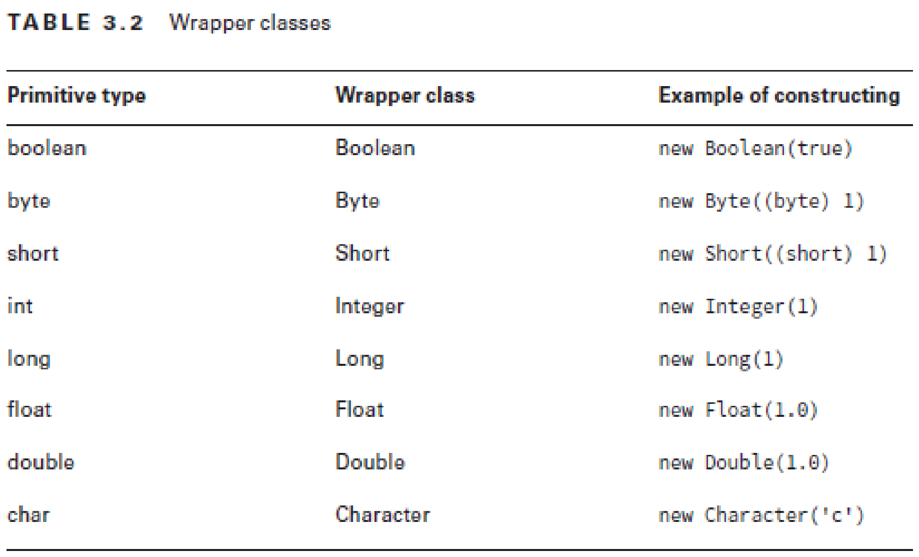


TABLE 3.2 Wrapper classes

Primitive type	Wrapper class	Example of constructing
boolean	Boolean	`new Boolean(true)`
byte	Byte	`new Byte((byte) 1)`
short	Short	`new Short((short) 1)`
int	Integer	`new Integer(1)`
long	Long	`new Long(1)`
float	Float	`new Float(1.0)`
double	Double	`new Double(1.0)`
char	Character	`new Character('c')`

The wrapper classes also have a method that converts back to a primitive. You don’t need to know much about the constructors or intValue() type methods for the exam because autoboxing has removed the need for them. There are also methods for converting a String to a primitive or wrapper class. You do need to know these methods. The parse methods, such as parseInt(), return a primitive, and the valueOf() method returns a wrapper class.
intValue() – Returns the primitive value of the wrapper class. Not required as autoboxing is introduced now.
parseInt() – Converts the string input to primitive
valueOf() – Converts the string input to Wrapper

```
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. If the String passed in is not valid for the given type, Java throws an exception. In these examples, letters and dots are not valid for an integer value:

```
int bad1 = Integer.parseInt("a"); // throws NumberFormatException
Integer bad2 = Integer.valueOf("123.45"); // throws NumberFormatException
```

The Character class doesn’t participate in the parse/valueOf methods. Since a String is made up of characters, you can just call charAt() normally.

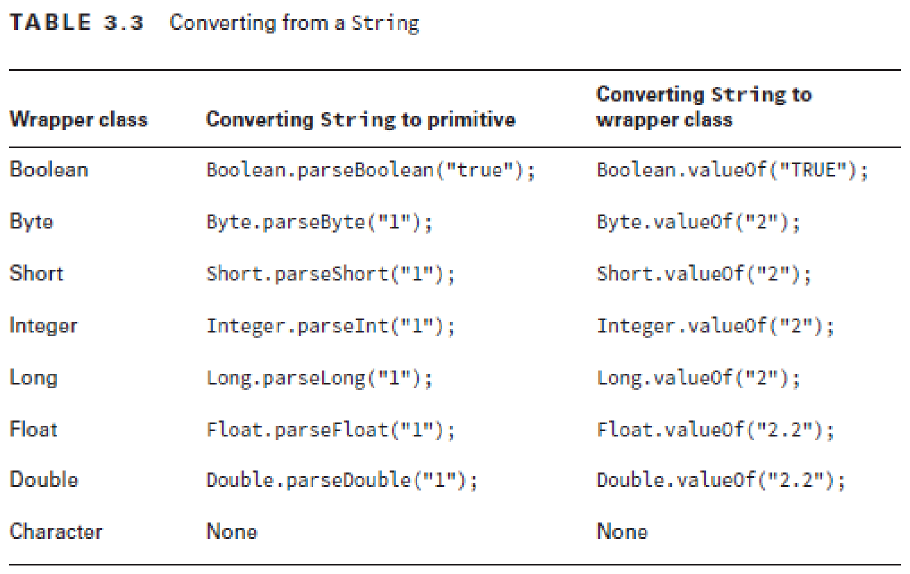


TABLE 3.3 Converting from a String

Wrapper class	Converting String to primitive	Converting String to wrapper class
Boolean	`Boolean.parseBoolean("true");`	`Boolean.valueOf("TRUE");`
Byte	`Byte.parseByte("1");`	`Byte.valueOf("2");`
Short	`Short.parseShort("1");`	`Short.valueOf("2");`
Integer	`Integer.parseInt("1");`	`Integer.valueOf("2");`
Long	`Long.parseLong("1");`	`Long.valueOf("2");`
Float	`Float.parseFloat("1");`	`Float.valueOf("2.2");`
Double	`Double.parseDouble("1");`	`Double.valueOf("2.2");`
Character	None	None

Since Java 5, you can just type the primitive value and Java will convert it to the relevant wrapper class for you. This is called *autoboxing*.

```
4: List<Double> weights = new ArrayList<>();
5: weights.add(50.5); // [50.5]
6: weights.add(new Double(60)); // [50.5, 60.0]
7: weights.remove(50.5); // [60.0]
8: double first = weights.get(0); // 60.0
```

Line 5 autoboxes the double primitive into a Double object and adds that to the List. Line 6 shows that you can still write code the long way and pass in a wrapper object. Line 7 again autoboxes into the wrapper object and passes it to remove(). Line 8 retrieves the Double and unboxes it into a double primitive. What do you think happens if you try to unbox a null?

```
3: List<Integer> heights = new ArrayList<>();
4: heights.add(null);
5: int h = heights.get(0); // NullPointerException
```

On line 4, we add a null to the list. This is legal because a null reference can be assigned to any reference variable. On line 5, we try to unbox that null to an int primitive. This is a problem. Java tries to get the int value of null. Since calling any method on null gives a NullPointerException, that is just what we get. Be careful when you see null in relation to autoboxing.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.remove(1);
System.out.println(numbers);
```

It actually outputs 1. After adding the two values, the List contains [1, 2]. We then request the element with index 1 be removed. That's right: index 1. Because there's already a remove() method that takes an int parameter, Java calls that method rather than autoboxing. If you want to remove the 2, you can write numbers.remove(new Integer(2)) to force wrapper class use.

Converting from an ArrayList to an Array

```
3: List<String> list = new ArrayList<>();
4: list.add("hawk");
5: list.add("robin");
6: Object[] objectArray = list.toArray();
7: System.out.println(objectArray.length); // 2
8: String[] stringArray = list.toArray(new String[0]);
9: System.out.println(stringArray.length); // 2
```

Line 6 shows that an ArrayList knows how to convert itself to an array. The only problem is that it defaults to an array of class Object. This isn’t usually what you want. Line 8 specifies the type of the array and does what we actually want. The advantage of specifying a size of 0 for the parameter is that Java will create a new array of the proper size for the return value. If you like, you can suggest a larger array to be used instead. If the ArrayList fits in that array, it will be returned. Otherwise, a new one will be created.

Converting from an array to a List is more interesting. The original array and created array backed List are linked. When a change is made to one, it is available in the other. It is a fixed-size list and is also known a *backed* List because the array changes with it. Pay careful attention to the values here:

```
20: String[] array = { "hawk", "robin" }; // [hawk, robin]
21: List<String> list = Arrays.asList(array); // returns fixed size list
22: System.out.println(list.size()); // 2
23: list.set(1, "test"); // [hawk, test]
24: array[0] = "new"; // [new, test]
25: for (String b : array) System.out.print(b + " "); // new test
26: list.remove(1); // throws UnsupportedOperation Exception
```

Line 21 converts the array to a List. Note that it isn’t the java.util.ArrayList we’ve grown used to. It is a fi xed-size, backed version of a List. Line 23 is okay because set() merely replaces an existing value. It updates both array and list because they point to the same data store. Line 24 also changes both array and list. Line 25 shows the array has changed to new test. Line 26 throws an exception because we are not allowed to change the size of the list.

This topic isn’t on the exam, but merging varargs with ArrayList conversion allows you to create an ArrayList in a cool way:

```
List<String> list = Arrays.asList("one", "two");
```

asList() takes varargs, which let you pass in an array or just type out the String values.This is handy when testing because you can easily create and populate a List on one line.

Sorting an ArrayList is very similar to sorting an array. You just use a different helper class:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(99);
numbers.add(5);
numbers.add(81);
Collections.sort(numbers);
```

```
System.out.println(numbers); [5, 81, 99]
```

As with an ArrayList, you need an import statement to work with the date and time classes. Most of them are in the java.time package. To use it, add this import to your program:

```
import java.time.*; // import time classes
```

```
System.out.println(LocalDate.now()); //2015-01-20
System.out.println(LocalTime.now()); //12:45:18.401
System.out.println(LocalDateTime.now()); //2015-01-20T12:45:18.401
```

Also, Java tends to use a 24-hour clock even though the United States uses a 12-hour clock with a.m./p.m.

```
LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);
LocalDate date2 = LocalDate.of(2015, 1, 20);
```

```
public static LocalDate of(int year, int month, int dayOfMonth)
public static LocalDate of(int year, Month month, int dayOfMonth)
```

Month is a special type of class called an enum.

Up to now, we've been like a broken record telling you that Java counts starting with 0. Well, months are an exception. For months in the new date and time methods, Java counts starting from 1 like we human beings do.

```
LocalTime time1 = LocalTime.of(6, 15); // hour and minute
LocalTime time2 = LocalTime.of(6, 15, 30); // + seconds
LocalTime time3 = LocalTime.of(6, 15, 30, 200); // + nanoseconds
```

```
public static LocalTime of(int hour, int minute)
public static LocalTime of(int hour, int minute, int second)
public static LocalTime of(int hour, int minute, int second, int nanos)
```

```
LocalDateTime dateTime1 = LocalDateTime.of(2015, Month.JANUARY, 20, 6, 15, 30);
LocalDateTime dateTime2 = LocalDateTime.of(date1, time1);
```

```
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second)
```

public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime)

Did you notice that we did not use a constructor in any of the examples? The date and time classes have private constructors to force you to use the static methods. The exam creators may try to throw something like this at you:
LocalDate d = new LocalDate(); // DOES NOT COMPILE
Don't fall for this. You are not allowed to construct a date or time object directly. Another trick is to see what happens when you pass invalid numbers to of(). For example:
LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException
You don't need to know the exact exception that's thrown, but it's a clear one:
java.time.DateTimeException: Invalid value for DayOfMonth (valid values 1 - 28/31): 32

Creating Dates in Java 7 and Earlier… There wasn't a way to specify just a date without the time. The Date class represented both the date and time whether you wanted it to or not. Trying to create a specific date required more code than it should have. Month indexes were 0 based instead of 1 based, which was confusing.

The date and time classes are immutable, just like String was. This means that we need to remember to assign the results of these methods to a reference variable so they are not lost.

```
12: LocalDate date = LocalDate.of(2014, Month.JANUARY, 20);
13: System.out.println(date); // 2014-01-20
14: date = date.plusDays(2);
15: System.out.println(date); // 2014-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date); // 2014-01-29
18: date = date.plusMonths(1);
19: System.out.println(date); // 2014-02-28
20: date = date.plusYears(5);
21: System.out.println(date); // 2019-02-28

22: LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
23: LocalTime time = LocalTime.of(5, 15);
24: LocalDateTime dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime); // 2020-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime); // 2020-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime); // 2020-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime); // 2020-01-18T19:14:30
```

It is common for date and time methods to be chained. For example, without the print statements, the previous example could be rewritten as follows:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);
LocalDateTime dateTime = LocalDateTime.of(date2,
time).minusDays(1).minusHours(10).minusSeconds(30);
```

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
date = date.plusMinutes(1); // DOES NOT COMPILE
```

LocalDate does not contain time. This means you cannot add minutes to it. This can be tricky in a chained sequence of additions/subtraction operations, so make sure you know which methods in Table 3.4 can be called on which of the three objects.

TABLE 3.4 Methods in `LocalDate`, `LocalTime`, and `LocalDateTime`

	Can call on `LocalDate`?	Can call on `LocalTime`?	Can call on `LocalDateTime`?
`plusYears/minusYears`	Yes	No	Yes
`plusMonths/minusMonths`	Yes	No	Yes
`plusWeeks/minusWeeks`	Yes	No	Yes
`plusDays/minusDays`	Yes	No	Yes
`plusHours/minusHours`	No	Yes	Yes
`plusMinutes/minusMinutes`	No	Yes	Yes
`plusSeconds/minusSeconds`	No	Yes	Yes
`plusNanos/minusNanos`	No	Yes	Yes

LocalDate and LocalDateTime have a method to convert them into long equivalents in relation to 1970. What's special about 1970? That's what UNIX started using for date standards, so Java reused it. And don't worry—you don't have to memorize the names for the exam.

- LocalDate has toEpochDay(), which is the number of days since January 1, 1970.
- LocalDateTime has toEpochTime(), which is the number of seconds since January 1, 1970.
- LocalTime does not have an epoch method. Since it represents a time that occurs on any date, it doesn't make sense to compare it in 1970. Although the exam pretends time zones don't exist, you may be wondering if this special January 1, 1970 is in a specifi c time zone. The answer is yes. This special time refers to when it was January 1, 1970 in GMT (Greenwich Mean Time). Greenwich is in England and GMT does not participate in daylight savings time. This makes it a good reference point. (Again, you don't have to know about GMT for the exam.)

There are five ways to create a Period class:

```
Period annually = Period.ofYears(1); // every 1 year
Period quarterly = Period.ofMonths(3); // every 3 months
Period everyThreeWeeks = Period.ofWeeks(3); // every 3 weeks
Period everyOtherDay = Period.ofDays(2); // every 2 days
Period everyYearAndAWeek = Period.of(1, 0, 7); // every year and 7 days
```

There's one catch. You cannot chain methods when creating a Period. The following code looks like it is equivalent to the everyYearAndAWeek example, but it's not. Only the last method is used because the Period.ofXXX methods are static methods.

```
Period wrong = Period.ofYears(1).ofWeeks(1); // every week
```

This tricky code is really like writing the following:

```
Period wrong = Period.ofYears(1);
wrong = Period.ofWeeks(7);
```

This is clearly not what you intended! That's why the of() method allows us to pass in the number of years, months, and days. They are all included in the same period. You will get a compiler warning about this. Compiler warnings tell you something is wrong or suspicious without failing compilation.

You've probably noticed by now that a Period is a day or more of time. There is also Duration, which is intended for smaller units of time. For Duration, you can specify the number of days, hours, minutes, seconds, or nanoseconds. And yes, you could pass 365 days to make a year, but you really shouldn't—that's what Period is for. Duration isn't on the exam since it roughly works the same way as Period. It's good to know it exists,though.

The last thing to know about Period is what objects it can be used with. Let's look at some code:

```
3: LocalDate date = LocalDate.of(2015, 1, 20);
4: LocalTime time = LocalTime.of(6, 15);
5: LocalDateTime dateTime = LocalDateTime.of(date, time);
6: Period period = Period.ofMonths(1);
7: System.out.println(date.plus(period)); // 2015-02-20
8: System.out.println(dateTime.plus(period)); // 2015-02-20T06:15
9: System.out.println(time.plus(period)); // UnsupportedTemporalTypeException
```

The date and time classes support many methods to get data out of them:

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
System.out.println(date.getDayOfWeek()); // MONDAY
System.out.println(date.getMonth()); // JANUARY
System.out.println(date.getYear()); // 2020
System.out.println(date.getDayOfYear()); // 20
```

We could use this information to display information about the date. However, it would be more work than necessary. Java provides a class called DateTimeFormatter to help us out. Unlike the LocalDateTime class, DateTimeFormatter can be used to format any type of date and/or time object. What changes is the format. DateTimeFormatter is in the package java.time.format.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
```

```
LocalDateTime dateTime = LocalDateTime.of(date, time);
System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
System.out.println(dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
```

ISO is a standard for dates. The output of the previous code looks like this:

```
2020-01-20
11:12:34
2020-01-20T11:12:34
```

This is a reasonable way for computers to communicate, but probably not how you want to output the date and time in your program. Luckily there are some predefined formats that are more useful:

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(shortDateTime.format(dateTime)); // 1/20/20
System.out.println(shortDateTime.format(date)); // 1/20/20
System.out.println(shortDateTime.format(time)); // UnsupportedTemporalTypeException
```

Here we say we want a localized formatter in the predefined short format. The last line throws an exception because a time cannot be formatted as a date. The format() method is declared on both the formatter objects and the date/time objects, allowing you to reference the objects in either order. The following statements print exactly the same thing as the previous code:

```
DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
System.out.println(dateTime.format(shortDateTime));
System.out.println(date.format(shortDateTime));
System.out.println(time.format(shortDateTime));
```

TABLE 3.5 `ofLocalized methods`

`DateTimeFormatter f = DateTime Formatter.____ (FormatStyle.SHORT);`	**Calling `f.format (localDate)`**	**Calling `f.format (localDateTime)`**	**Calling `f.format (localTime)`**
`ofLocalizedDate`	Legal – shows whole object	Legal – shows just date part	Throws runtime exception
`ofLocalizedDateTime`	Throws runtime exception	Legal – shows whole object	Throws runtime exception
`ofLocalizedTime`	Throws runtime exception	Legal – shows just time part	Legal – shows whole object

There are two predefined formats that can show up on the exam: SHORT and MEDIUM. The other predefined formats involve time zones, which are not on the exam.

```
LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);
LocalTime time = LocalTime.of(11, 12, 34);
LocalDateTime dateTime = LocalDateTime.of(date, time);
DateTimeFormatter shortF = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
DateTimeFormatter mediumF =
DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(shortF.format(dateTime)); // 1/20/20 11:12 AM
System.out.println(mediumF.format(dateTime)); // Jan 20, 2020 11:12:34 AM
```

If you don’t want to use one of the predefined formats, you can create your own. For example, this code spells out the month:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");
System.out.println(dateTime.format(f)); // January 20, 2020, 11:12
```

MMMM M represents the month. The more Ms you have, the more verbose the Java output. For example, M outputs 1, MM outputs 01, MMM outputs Jan, and MMMM outputs January.
dd d represents the date in the month. As with month, the more ds you have, the more verbose the Java output. dd means to include the leading zero for a single-digit month.
, Use **,** if you want to output a comma (this also appears after the year).
yyyy y represents the year. yy outputs a two-digit year and yyyy outputs a four-digit year.
hh h represents the hour. Use hh to include the leading zero if you’re outputting a single digit hour.
: Use : if you want to output a colon.
mm m represents the minute.

Now that you know how to convert a date or time to a formatted String, you’ll find it easy to convert a String to a date or time. Just like the format() method, the parse() method takes a formatter as well. If you don’t specify one, it uses the default for that type.

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");
LocalDate date = LocalDate.parse("01 02 2015", f);
LocalTime time = LocalTime.parse("11:22");
System.out.println(date); // 2015-01-02
System.out.println(time); // 11:22
```

Here we show using both a custom formatter and a default value. This isn’t common, but you might have to read code that looks like this on the exam. Parsing is consistent in that if anything goes wrong, Java throws a runtime exception. That could be a format that doesn’t match the String to be parsed or an invalid date.

Methods and Encapsulation

Java methods start with an access modifier of public, private, protected or blank (default access). This is followed by an optional specifier such as static, final, or abstract. Next comes the return type, which is void or a Java type. The method name follows, using standard Java identifier rules. Zero or more parameters go in parentheses as the parameter list. Next come any optional exception types. Finally, zero or more statements go in braces to make up the method body.

Using the private keyword means the code is only available from within the same class. Default (package private) access means the code is only available from within the same package. Using the protected keyword means the code is available from the same package or subclasses. Using the public keyword means the code is available from anywhere. Static methods and static variables are shared by the class. When referenced from outside the class, they are called using the classname—for example, StaticClass.method(). Instance members are allowed to call static members, but static members are not allowed to call instance members. Static imports are used to import static members.

Java uses pass-by-value, which means that calls to methods create a copy of the parameters. Assigning new values to those parameters in the method doesn't affect the caller's variables. Calling methods on objects that are method parameters changes the state of those objects and is reflected in the caller.

Overloaded methods are methods with the same name but a different parameter list. Java calls the most specific method it can find. Exact matches are preferred, followed by wider primitives. After that comes autoboxing and finally varargs.

Constructors are used to instantiate new objects. The default no-argument constructor is called when no constructor is coded. Multiple constructors are allowed and can call each other by writing this(). If this() is present, it must be the first statement in the constructor. Constructors can refer to instance variables by writing this before a variable name to indicate they want the instance variable and not the method parameter with that name. The order of initialization is the superclass (which we will cover in Chapter 5); static variables and static initializers in the order they appear; instance variables and instance initializers in the order they appear; and finally the constructor.

Encapsulation refers to preventing callers from changing the instance variables directly. This is done by making instance variables private and getters/setters public. Immutability refers to preventing callers from changing the instance variables at all. This uses several techniques, including removing setters. JavaBeans use methods beginning with is and get for boolean and non-boolean property types, respectively. Methods beginning with set are used for setters.

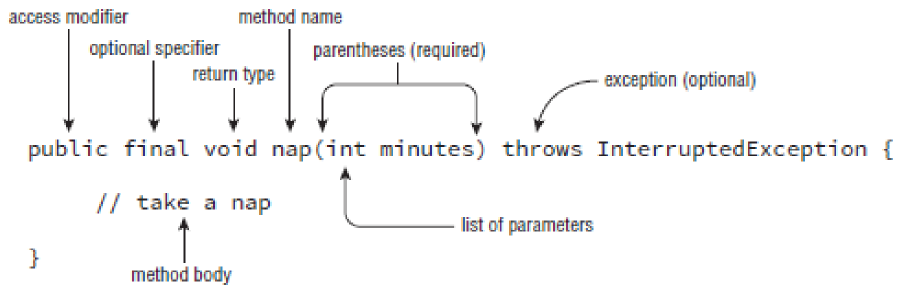
Lambda expressions, or lambdas, allow passing around blocks of code. The full syntax looks like (String a, String b) -> { return a.equals(b); }. The parameter types can be omitted. When only one parameter is specified without a type, the parentheses can also be omitted. The braces and return statement can be omitted for a single statement, making the short form (a -> a.equals(b). Lambdas are passed to a method expecting an interface with one method. Predicate is a common interface. It has one method named test that returns a boolean and takes any type. The removeIf() method on ArrayList takes a Predicate.

A sample method signature is public static void method(String... args) throws Exception {}.

Static imports import static members. They are written as import static, not *static import*. Make sure they are importing static methods or variables rather than classnames.

Order of initialization is the superclass, static variables/initializers, instance variables/initializers, and the constructor.

FIGURE 4.1 Method signature



```
public final void nap(int minutes) throws InterruptedException {
    // take a nap
}
```

Java offers four choices of access modifier:
public The method can be called from any class.
private The method can only be called from within the same class.
protected The method can only be called from classes in the same package or subclasses.
Default (Package Private) Access The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

There's a default keyword in Java. You saw it in the switch statement and you'll see it again in the interfaces. It's not used for access control.

There are a number of optional specifiers, but most of them aren't on the exam. Optional specifiers come from the following list. Unlike with access modifiers, you can have multiple specifiers in the same method (although not all combinations are legal). When this happens, you can specify them in any order. And since it is optional, you can't have any of them at all. This means you can have zero or more specifiers in a method declaration.
static Used for class methods.
abstract Used when not providing a method body.
final Used when a method is not allowed to be overridden by a subclass.

synchronized On the OCP but not the OCA exam.
native Not on the OCA or OCP exam. Used when interacting with code written in another language such as C++.
strictfp Not on the OCA or OCP exam. Used for making floating-point calculations portable.

```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {} // DOES NOT COMPILE
public void final walk6() {} // DOES NOT COMPILE
final public void walk7() {}
```

Remember that a method must have a return type. If no value is returned, the return type is void. You cannot omit the return type.

Methods that have a return type of void are permitted to have a return statement with no value returned or omit the return statement entirely.

```
public void walk1() { }
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() { } // DOES NOT COMPILE
public walk5() { } // DOES NOT COMPILE
String walk6(int a) { if (a == 4) return ""; } // DOES NOT COMPILE
```

walk6() is a little tricky. There is a return statement, but it doesn't always get run. If a is 6, the return statement doesn't get executed. Since the String always needs to be returned, the compiler complains.

```
int integer() {
return 9;
}
int long() {
return 9L; // DOES NOT COMPILE
}
```

Method names follow the same rules as with variable names. To review, an identifier may only contain letters, numbers, $, or _. Also, the first character is not allowed to be a number, and reserved words are not allowed. By convention, methods begin with a lowercase letter but are not required to.

```
public void walk1() { }
public void 2walk() { } // DOES NOT COMPILE
public walk3 void() { } // DOES NOT COMPILE
public void Walk_$() { }
public void() { } // DOES NOT COMPILE
```

Although the parameter list is required, it doesn't have to contain any parameters. This means you can just have an empty pair of parentheses after the method name, such as void nap(){}. If you do have multiple parameters, you separate them with a comma.

```
public void walk2 { } // DOES NOT COMPILE
public void walk4(int a; int b) { } // DOES NOT COMPILE
```

```
public void zeroExceptions() { }
public void oneException() throws IllegalArgumentException { }
public void twoExceptions() throws
IllegalArgumentException, InterruptedException { }
```

You might be wondering what methods do with these exceptions. The calling method can throw the same exceptions or handle them.

A method body is simply a code block. It has braces that contain zero or more Java statements.

```
public void walk1() { }
public void walk2; // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

A method may use a vararg parameter (variable argument) as if it is an array. It is a little different than an array, though. A vararg parameter must be the last element in a method's parameter list. This implies you are only allowed to have one vararg parameter per method.

```
public void walk1(int... nums) { }
public void walk2(int start, int... nums) { }
public void walk3(int... nums, int start) { } // DOES NOT COMPILE
public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
```

When calling a method with a vararg parameter, you have a choice. You can pass in an array, or you can list the elements of the array and let Java create it for you. You can even omit the vararg values in the method call and Java will create an array of length zero for you.

```
public static void walk(int start, int... nums) {
System.out.println(nums.length);
}
public static void main(String[] args) {
walk(1); // 0
walk(1, 2); // 1
walk(1, 2, 3); // 2
walk(1, new int[] {4, 5}); // 2
}
```

You've seen that Java will create an empty array if no parameters are passed for a vararg. However, it is still possible to pass null explicitly:

```
walk(1, null); // throws a NullPointerException
```

Since null isn't an int, Java treats it as an array reference that happens to be null. It just passes on the null array object to walk. Then the walk() method throws an exception because it tries to determine the length of null. Accessing a vararg parameter is also just like accessing an array. It uses array indexing.

```
package pond.swan;
import pond.shore.Bird; // in different package than Bird
public class Swan extends Bird { // but subclass of bird
public void swim() {
floatInWater(); // package access to superclass
System.out.println(text); // package access to superclass
}
public void helpOtherSwanSwim() {
Swan other = new Swan();
other.floatInWater(); // package access to superclass
System.out.println(other.text);// package access to superclass
}
public void helpOtherBirdSwim() {
Bird other = new Bird();
other.floatInWater(); // DOES NOT COMPILE
System.out.println(other.text); // DOES NOT COMPILE
}
}
```

```
package pond.goose;
import pond.shore.Bird;
public class Goose extends Bird {
public void helpGooseSwim() {
Goose other = new Goose();
other.floatInWater();
System.out.println(other.text);
}
public void helpOtherGooseSwim() {
Bird other = new Goose();
other.floatInWater(); // DOES NOT COMPILE
System.out.println(other.text); // DOES NOT COMPILE
} }
```

```
package pond.duck;
import pond.goose.Goose;
public class GooseWatcher {
public void watch() {
Goose goose = new Goose();
goose.floatInWater(); // DOES NOT COMPILE
```

} }
This code doesn't compile because we are not in the Goose class. The floatInWater() method is declared in Bird. GooseWatcher is not in the same package as Bird, nor does it extend Bird. Goose extends Bird. That only lets Goose refer to floatInWater() and not callers of Goose.

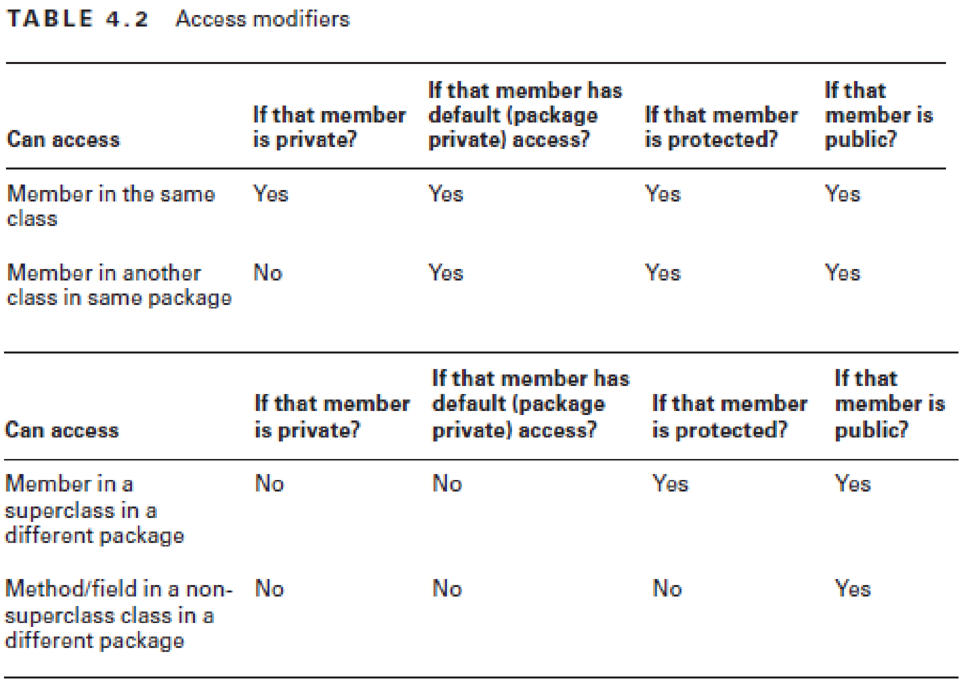


TABLE 4.2 Access modifiers

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in the same class	Yes	Yes	Yes	Yes
Member in another class in same package	No	Yes	Yes	Yes
Member in a superclass in a different package	No	No	Yes	Yes
Method/field in a non-superclass class in a different package	No	No	No	Yes

Except for the main() method, we've been looking at instance methods. Static methods don't require an instance of the class. They are shared among all users of the class. You can think of statics as being a member of the single class object that exist independently of any instances of that class.

Each class has a copy of the instance variables. There is only one copy of the code for the instance methods. Each instance of the class can call it as many times as it would like. However, each call of an instance method (or any method) gets space on the stack for method parameters and local variables. The same thing happens for static methods. There is one copy of the code. Parameters and local variables go on the stack. Just remember that only data gets its "own copy." There is no need to duplicate copies of the code itself.

We said that the JVM basically calls Koala.main() to get the program started. You can do this too. We can have a KoalaTester that does nothing but call the main() method.

```
public class KoalaTester {
public static void main(String[] args) {
Koala.main(new String[0]); // call static method
}
}
```

You can use an instance of the object to call a static method. The compiler checks for the type of the reference and uses that instead of the object—which is sneaky of Java. This code is perfectly legal:

```
5: Koala k = new Koala();
6: System.out.println(k.count); // k is a Koala
7: k = null;
8: System.out.println(k.count); // k is still a Koala
```

Believe it or not, this code outputs 0 twice. Line 6 sees that k is a Koala and count is a static variable, so it reads that static variable. Line 8 does the same thing. Java doesn't care that k happens to be null. Since we are looking for a static, it doesn't matter.

```
Koala.count = 4;
Koala koala1 = new Koala();
Koala koala2 = new Koala();
koala1.count = 6;
koala2.count = 5;
System.out.println(Koala.count);
```

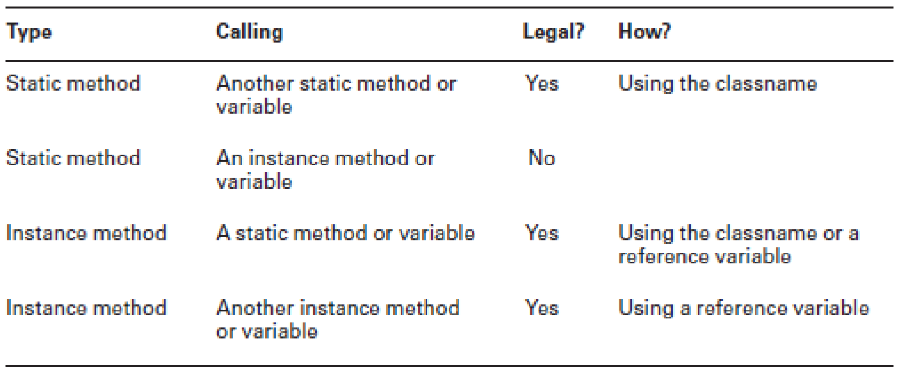
Hopefully, you answered 5. There is only one count variable since it is static. It is set to 4, then 6, and finally winds up as 5. All the Koala variables are just distractions.

A static member cannot call an instance member. This shouldn't be a surprise since static doesn't require any instances of the class to be around.

```
public class Static {
private String name = "Static class";
public static void first() { }
public static void second() { }
public void third() { System.out.println(name); }
public static void main(String args[]) {
first();
second();
third(); // DOES NOT COMPILE
} }
```

A static method or instance method can call a static method because static methods don't require an object to use. Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object. Similar logic applies for the instance and static variables.

TABLE 4.3 Static vs. instance calls



Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

A common use for static variables is counting the number of instances.

```
public static void main(String[] args) {
Counter c1 = new Counter();
Counter c2 = new Counter();
Counter c3 = new Counter();
System.out.println(count); // 3
}
}
```

Other static variables are meant to never change during the program. This type of variable is known as a *constant*. It uses the final modifier to ensure the variable never changes. static final constants use a different naming convention than other variables. They use all uppercase letters with underscores between "words."

Static initializers look similar to instance initializers. They add the static keyword to specify they should be run when the class is first used. For example:

```
private static final int NUM_SECONDS_PER_HOUR;
static {
int numSecondsPerMinute = 60;
int numMinutesPerHour = 60;
NUM_SECONDS_PER_HOUR = numSecondsPerMinute * numMinutesPerHour;
}
```

The static initializer runs when the class is first used. The statements in it run and assign any static variables as needed. There is something interesting about this example. We just got through saying that final variables aren't allowed to be reassigned. The key here is that the static initializer is the first assignment. And since it occurs up front, it is okay.

There is another type of import called a static import. Regular imports are for importing classes. Static imports are for importing static members of classes. Just like regular imports, you can use a wildcard or import a specific member. The idea is that you shouldn't have to specify where each static method or variable comes from each time you use it. An example of when static imports shine are when you are referring to a lot of constants in another class.

An interesting case is what would happen if we created same method in our class. Java would give it preference over the imported one and the method we coded would be used.

```
1: import static java.util.Arrays; // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*; // DOES NOT COMPILE
4: public class BadStaticImports {
5: public static void main(String[] args) {
6: Arrays.asList("one"); // DOES NOT COMPILE
7: } }
```

Line 6 is sneaky. We imported the asList method on line 2. However we did not import the Arrays class anywhere. This makes it okay to write asList("one"); but not Arrays.asList("one");. There's only one more scenario with static imports. In Chapter 1, you learned that importing two classes with the same name gives a compiler error. This is true of static imports as well. The compiler will complain if you try to explicitly do a static import of two methods with the same name or two static variables with the same name.

Java is a "pass-by-value" language. This means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller.

```
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }
```

As you can see, we can overload by changing anything in the parameter list. We can have a different type, more types, or the same types in a different order. Also notice that the access modifier and exception list are irrelevant to overloading.

```
public void fly(int numMiles) { }
public int fly(int numMiles) { } // DOES NOT COMPILE
```

This method doesn't compile because it only differs from the original by return type. The parameter lists are the same so they are duplicate methods as far as Java is concerned.

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { } // DOES NOT COMPILE
```

It shouldn't be a surprise that you can call either method by passing an array:

```
fly(new int[] { 1, 2, 3 });
```

However, you can only call the varargs version with stand-alone parameters:

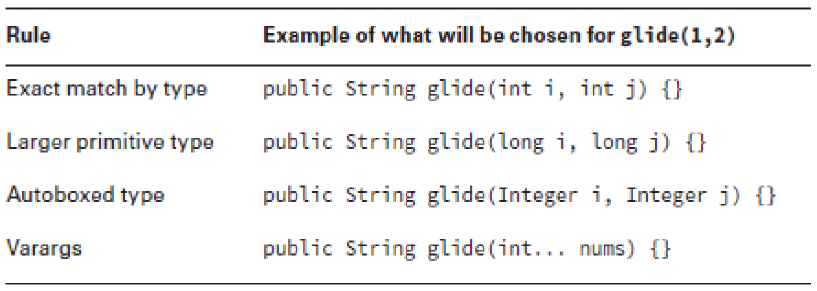
```
fly(1, 2, 3);
```

Obviously, this means they don't compile exactly the same. The parameter list is the same, though.

```
public class ReferenceTypes {
public void fly(String s) {
System.out.print("string ");
}
public void fly(Object o) {
System.out.print("object ");
}
public static void main(String[] args) {
ReferenceTypes r = new ReferenceTypes();
r.fly("test");
r.fly(56);
} }
```

The answer is "string object". The first call is a String and finds a direct match. There's no reason to use the Object version when there is a nice String parameter list just waiting to be called. The second call looks for an int parameter list. When it doesn't find one, it autoboxes to Integer. Since it still doesn't find a match, it goes to the Object one.

TABLE 4.4 Order Java uses to choose the right overloaded method



Rule	Example of what will be chosen for `glide(1,2)`
Exact match by type	`public String glide(int i, int j) {}`
Larger primitive type	`public String glide(long i, long j) {}`
Autoboxed type	`public String glide(Integer i, Integer j) {}`
Varargs	`public String glide(int... nums) {}`

As accommodating as Java is with trying to find a match, it will only do one conversion:

```
public class TooManyConversions {
public static void play(Long l) { }
public static void play(Long... l) { }
public static void main(String[] args) {
play(4); // DOES NOT COMPILE
play(4L); // calls the Long version
} }
```

Here we have a problem. Java is happy to convert the int 4 to a long 4 or an Integer 4. It cannot handle converting in two steps to a long and then to a Long. If we had public static void play(Object o) { }, it would match because only one conversion would be necessary: from int to Integer. An Integer is an Object.

We keep saying *generated*. This happens during the compile step. If you look at the file with the .java extension, the constructor will still be missing. It is only in the compiled file with the.class extension that it makes an appearance. Remember that a default constructor is only supplied if there are no constructors
present.

Having a private constructor in a class tells the compiler not to provide a default noargument constructor. It also prevents other classes from instantiating the class. This is useful when a class only has static methods or the class wants to control all calls to create new instances of itself.

You can have multiple constructors in the same class as long as they have different method signatures. When overloading methods, the method name and parameter list needed to match. With constructors, the name is always the same since it has to be the same as the name of the class. This means constructors must have different parameters in order to be overloaded.

this() has one special rule you need to know. If you choose to call it, the this() call must be the first noncommented statement in the constructor.

```
public Hamster(int weight) {
System.out.println("in constructor");
// ready to call this
this(weight, "brown"); // DOES NOT COMPILE
}
```

Overloaded constructors often call each other. One common technique is to have each constructor add one parameter until getting to the constructor that does all the work. This approach is called *constructor chaining*.

As you saw earlier in the chapter, final instance variables must be assigned a value exactly once. We saw this happen in the line of the declaration and in an instance initializer. There is one more location this assignment can be done: in the constructor. The constructor is part of the initialization process, so it is allowed to assign final instance variables in it. By the time the constructor completes, all final instance variables must have been set.

Order of initialization

- If there is a superclass, initialize it first (we'll cover this rule in the next chapter. For now, just say "no superclass" and go on to the next rule.)
- Static variable declarations and static initializers in the order they appear in the file.
- Instance variable declarations and instance initializers in the order they appear in the file.
- The constructor.

Keep in mind that the four rules apply only if an object is instantiated. If the class is referred to without a new call, only rules 1 and 2 apply. The other two rules relate to instances and constructors. They have to wait until there is code to instantiate the object.

```
public class InitializationOrder {
private String name = "Torchie";
{ System.out.println(name); }
private static int COUNT = 0;
static { System.out.println(COUNT); }
{ COUNT++; System.out.println(COUNT); }
public InitializationOrder() {
System.out.println("constructor");
}
public static void main(String[] args) {
System.out.println("read to construct");
new InitializationOrder();
}
}
```

The output looks like this:

```
0
read to construct
Torchie
1
constructor
```

```
public class YetMoreInitializationOrder {
static { add(2); }
static void add(int num) { System.out.print(num + " "); }
YetMoreInitializationOrder() { add(5); }
static { add(4); }
{ add(6); }
static { new YetMoreInitializationOrder(); }
{ add(8); }
public static void main(String[] args) { } }
```

The correct answer is 2 4 6 8 5.

Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables. Callers are required to use these methods. Let’s take a look at our newly encapsulated Swan class:

```
public class Swan {
private int numberEggs; // private
public int getNumberEggs() { // getter
return numberEggs;
}
public void setNumberEggs(int numberEggs) { // setter
if (numberEggs >= 0) // guard condition
this.numberEggs = numberEggs;
} }
```

We added a method on lines 3–5 to read the value, which is called an accessor method or a getter. We also added a method on lines 6–9 to update the value, which is called a mutator method or a setter.

For encapsulation, remember that data (an instance variable) is private and getters/setters are public. Java defines a naming convention that is used in *JavaBeans*. JavaBeans are reusable software components. JavaBeans call an instance variable a *property*.

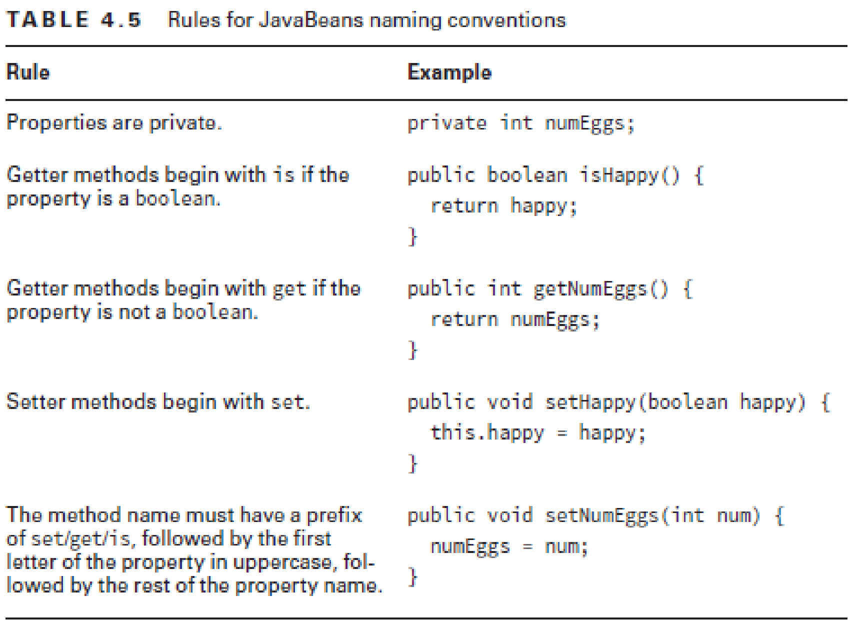


TABLE 4.5 Rules for JavaBeans naming conventions

Rule	Example
Properties are private.	`private int numEggs;`

Getter methods begin with `is` if the property is a `boolean`.	`public boolean isHappy() {`
	`  return happy;`
	`}`

Getter methods begin with `get` if the property is not a `boolean`.	`public int getNumEggs() {`
	`  return numEggs;`
	`}`

Setter methods begin with `set`.	`public void setHappy(boolean happy) {`
	`  this.happy = happy;`
	`}`

The method name must have a prefix of set/get/`is`, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) {`
	`  numEggs = num;`
	`}`

Encapsulating data is helpful because it prevents callers from making uncontrolled changes to your class. Another common technique is making classes immutable so they cannot be changed at all.

One step in making a class immutable is to omit the setters. But wait: we still want the caller to be able to specify the initial value—we just don't want it to change after the object is created. Constructors to the rescue!
Remember, immutable is only measured after the object is constructed. Immutable classes are allowed to have values. They just can't change after instantiation.

When you are writing an immutable class, be careful about the return types. On the surface, this class appears to be immutable since there is no setter:

```
public class NotImmutable {
private StringBuilder builder;
public NotImmutable(StringBuilder b) {
builder = b;
}
public StringBuilder getBuilder() {
return builder;
} }
```
The problem is that it isn't' really.
```
StringBuilder sb = new StringBuilder("initial");
NotImmutable problem = new NotImmutable(sb);
sb.append(" added");
StringBuilder gotBuilder = problem.getBuilder();
gotBuilder.append(" more");
System.out.println(problem.getBuilder());
```
This outputs "initial added more"—clearly not what we were intending. The problem is that we are just passing the same StringBuilder all over. The caller has a reference since it was passed to the constructor. Anyone who calls the getter gets a reference too. A solution is to make a copy of the mutable object. This is called a defensive copy.
```
public Mutable(StringBuilder b) {
builder = new StringBuilder(b);
}
public StringBuilder getBuilder() {
return new StringBuilder(builder);
}
```
Now the caller can make changes to the initial sb object and it is fi ne. Mutable no longer cares about that object after the constructor gets run. The same goes for the getter: callers can change their StringBuilder without affecting Mutable. Another approach for the getter is to return an immutable object:
```
public String getValue() {
return builder.toString();
}
```
There's no rule that says we have to return the same type as we are storing. String is safe to return because it is immutable in the first place. To review, encapsulation refers to preventing callers from changing the instance variables directly. Immutability refers to preventing callers from changing the instance variables at all.

Java is an object-oriented language at heart. You've seen plenty of objects by now. In Java 8, the language added the ability to write code using another style. *Functional programming* is a way of writing code more declaratively. You specify what you want to do rather than dealing with the state of objects. You focus more on expressions than loops.

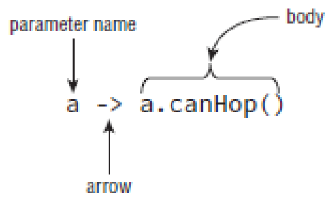
Functional programming uses lambda expressions to write code. A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an anonymous method. It has parameters and a body just like full-fledged methods do, but it doesn't have a name like a real method. Lambda expressions are often referred to as lambdas for short. You might also know them as closures if Java isn't your first language. If you had a bad experience with closures in the past, don't worry. They are far simpler in Java.

In other words, a lambda expression is like a method that you can pass as if it were a variable. This code uses a concept called deferred execution. *Deferred execution* means that code is specified now but will run later the
method calls it.

The syntax of lambdas is tricky because many parts are optional. These two lines do the exact same thing:
a -> a.canHop()
(Animal a) -> { return a.canHop(); }
Let's look at what is going on here. The first example has three parts:

- Specify a single parameter with the name a
- The arrow operator to separate the parameter and body
- A body that calls a single method and returns the result of that method

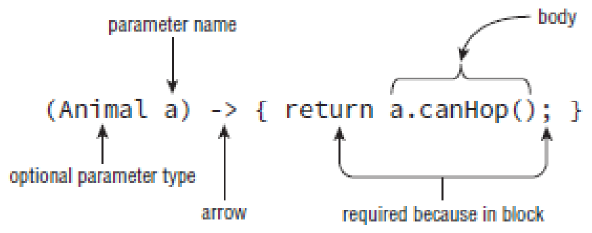
FIGURE 4.5 Lambda syntax omitting optional parts



The second example also has three parts; it's just more verbose

- Specify a single parameter with the name a and stating the type is Animal
- The arrow operator to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a return statement

FIGURE 4.6 Lambda syntax, including optional parts


parameter name
body
(Animal a) -> { return a.canHop(); }
optional parameter type
arrow
required because in block

The parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do as little typing as possible.

It shouldn't be news to you that we can omit braces when we only have a single statement. We did this with if statements and loops already. What is different here is that the rules change when you omit the braces. Java doesn't require you to type return or use a semicolon when no braces are used. This special shortcut doesn't work when we have two or more statements. At least this is consistent with using {} to create blocks of code elsewhere.

```
3: print(() -> true); // 0 parameters
4: print(a -> a.startsWith("test")); // 1 parameter
5: print((String a) -> a.startsWith("test")); // 1 parameter
6: print((a, b) -> a.startsWith("test")); // 2 parameters
7: print((String a, String b) -> a.startsWith("test")); // 2 parameters
```

```
print(a, b -> a.startsWith("test")); // DOES NOT COMPILE
print(a -> { a.startsWith("test"); }); // DOES NOT COMPILE
print(a -> { return a.startsWith("test") }); // DOES NOT COMPILE
```
The first line needs parentheses around the parameter list. Remember that the parentheses are *only* optional when there is one parameter and it doesn't have a type declared. The second line is missing the return keyword. The last line is missing the semicolon.

Lambdas are allowed to access variables. Here's an example:
```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```
The trick is that they cannot access all variables. Instance and static variables are okay. Method parameters and local variables are fine if they are not assigned new values.

There is one more issue you might see with lambdas. We've been defining an argument list in our lambda expressions. Since Java doesn't allow us to redeclare a local variable, the following is an issue:
```
(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE
```
We tried to redeclare a, which is not allowed. By contrast, the following line is okay because it uses a different variable name:
```
(a, b) -> { int c = 0; return 5;}
```

Lambdas work with interfaces that have only one method. These are called functional interfaces—interfaces that can be used with functional programming. (It's actually more complicated than this, but for the OCA exam this definition is fine.) You can imagine that we'd have to create lots of interfaces like this to use lambdas. We want to test Animals and Strings and Plants and anything else that we come across. Luckily, Java recognizes that this is a common problem and provides such an interface for us. It's in the package java.util.function and the gist of it is as follows:
```
public interface Predicate<T> {
boolean test(T t);
}
```
That looks a lot like our method. The only difference is that it uses this type T instead of Animal. That's the syntax for generics. It's like when we created an ArrayList and got to specify any type that goes in it. This means we don't need our own interface anymore and can put everything related to our search in one class.

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4: public static void main(String[] args) {
5: List<Animal> animals = new ArrayList<Animal>();
6: animals.add(new Animal("fish", false, true));
7:
8: print(animals, a -> a.canHop());
9: }
10: private static void print(List<Animal> animals, Predicate<Animal> checker) {
11: for (Animal animal : animals) {
12: if (checker.test(animal))
13: System.out.print(animal + " ");
14: }
15: System.out.println();
16: }
17: }
```

This time, line 10 is the only one that changed. We expect to have a Predicate passed in that uses type Animal. Pretty cool. We can just use it without having to write extra code.

Java 8 even integrated the Predicate interface into some existing classes. There is only one you need to know for the exam. ArrayList declares a removeIf() method that takes a Predicate.

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies); // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies); // [hoppy]
```

Class Design

Java classes follow a multilevel single-inheritance pattern in which every class has exactly one direct parent class, with all classes eventually inheriting from java.lang.Object. Java interfaces simulate a limited form of multiple inheritance, since Java classes may implement multiple interfaces.

By definition, an abstract type cannot be instantiated directly and requires a concrete subclass for the code to be used. Default and static interface methods are new to Java 8.

A Java class that extends another class inherits all of its public and protected methods and variables. The first line of every constructor is a call to another constructor within the class using this() or a call to a constructor of the parent class using the super() call. If the parent class doesn't contain a noargument constructor, an explicit call to the parent constructor must be provided. Parent methods and objects can be accessed explicitly using the super keyword. Finally, all classes in Java extend java.lang.Object either directly or from a superclass.

The Java compiler allows methods to be overridden in subclasses if certain rules are followed: a method must have the same signature, be at least as accessible as the parent method, must not declare any new or broader exceptions, and must use covariant return types.

When a static method is recreated in a subclass, it is referred to as method hiding. Likewise, variable hiding is when a variable name is reused in a subclass. In both situations, the original method or variable still exists and is used in methods that reference the object in the parent class. For method hiding, the use of static in the method declaration must be the same between the parent and child class. Finally, variable and method hiding should generally be avoided since it leads to confusing and difficult-to-follow code.

Both method overloading and overriding involve creating a new method with the same name as an existing method. When the method signature is the same, it is referred to as method overriding and must follow a specific set of override rules to compile. When the method signature is different, with the method taking different inputs, it is referred to as method overloading and none of the override rules are required.

In Java, classes and methods can be declared as abstract. Abstract classes cannot be instantiated and require a concrete subclass to be accessed. Abstract classes can include any number, including zero, of abstract and nonabstract methods. Abstract methods follow all the method override rules and may only be defined within abstract classes. The first concrete subclass of an abstract class must implement all the inherited methods. Abstract classes and methods may not be marked as final or private.

Interfaces are similar to a specialized abstract class in which only abstract methods and constant static final variables are allowed. New to Java 8, an interface can also define default and static methods with method bodies. All members of an interface are assumed to be public. Methods are assumed to be abstract if not explicitly marked as default or static. An interface that extends another interface inherits all its abstract methods. An interface cannot extend a class, nor can a class extend an interface. Finally, classes may implement any number of interfaces.

A default method allows a developer to add a new method to an interface used in existing implementations, without forcing other developers using the interface to recompile their code. A developer using the interface may override the default method or use the provided one. A static method in an interface follows the same rules for a static method in a class.

An object in Java may take on a variety of forms, in part depending on the reference used to access the object. Methods that are overridden will be replaced everywhere they are used, whereas methods and variables that are hidden will only be replaced in the classes and subclasses that they are defined. It is common to rely on polymorphic parameters—the ability of methods to be automatically passed as a superclass or interface reference—when creating method definitions.

An instance can be automatically cast to a superclass or interface reference without an explicit cast. Alternatively, an explicit cast is required if the reference is being narrowed to a subclass of the object. The Java compiler doesn't permit casting to unrelated types. You should be able to discern between compiler-time casting errors and those that will not occur until runtime and that throw a CastClassException.

We refer to any class that inherits from another class as a *child class*, or a descendent of that class. Alternatively, we refer to the class that the child inherits from as the *parent class*, or an ancestor of the class.

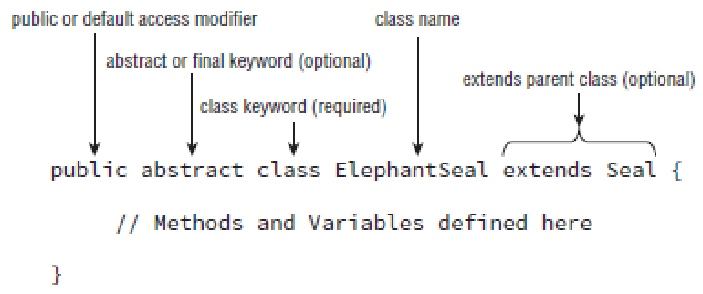
Java supports *single inheritance*, by which a class may inherit from only one direct parent class. Java also supports multiple levels of inheritance, by which one class may extend another class, which in turn extends another class. You can extend a class any number of times, allowing each descendent to gain access to its ancestor's members.

By design, Java doesn't support multiple inheritance in the language because studies have shown that multiple inheritance can lead to complex, often difficult-to-maintain code. Java does allow one exception to the single inheritance rule: classes may implement multiple interfaces.

If you have an object or method defined in all of the parents, which one does the child inherit? There is no natural ordering for parents in this example, which is why Java avoids these issues by disallowing multiple inheritance altogether.

It is possible in Java to prevent a class from being extended by marking the class with the final modifier.

FIGURE 5.2 Defining and extending a class


public or default access modifier
abstract or final keyword (optional)
class keyword (required)
class name
extends parent class (optional)

```
public abstract class ElephantSeal extends Seal {

     // Methods and Variables defined here

}
```

For the OCA exam, you should only be familiar with public and default package-level class access modifiers, because these are the only ones that can be applied to top-level classes within a Java file. The protected and private modifiers can only be applied to inner classes, which are classes that are defined within other classes, but this is well out of scope for the OCA exam.

The public access modifier applied to a class indicates that it can be referenced and used in any class. The default package private modifier, which is the lack of any access modifier, indicates the class can be accessed only by a subclass or class within the same package.

As you know, a Java fi le can have many classes but at most one public class. In fact, it may have no public class at all. One feature of using the default package private modifier is that you can define many classes within the same Java fi le.

The rules for applying class access modifiers are identical for interfaces. There can be at most one public class or interface in a Java fi le. Like classes, top-level interfaces can also be declared with the public or default modifiers.

In Java, all classes inherit from a single class, java.lang.Object. Furthermore, java.lang.Object is the only class that doesn't have any parent classes. You might be wondering, "None of the classes I've written so far extend java.lang.Object, so how do all classes inherit from it?" The answer is that the compiler has been automatically inserting code into any class you write that doesn't extend a specific class.

The key is that when Java sees you define a class that doesn't extend another class, it immediately adds the syntax extends java.lang.Object to the class definition. If you define a new class that extends an existing class, Java doesn't add this syntax, although the new class still inherits from java.lang.Object.

In Java, the first statement of every constructor is either a call to another constructor within the class, using this(), or a call to a constructor in the direct parent class, using super(). If a parent constructor takes arguments, the super constructor would also take arguments.

Following three class and constructor definitions are equivalent, because the compiler will automatically convert
them all to the last example:

```
public class Donkey {
}
public class Donkey {
public Donkey() {
}
}
public class Donkey {
public Donkey() {
super();
}
}
```

What happens if the parent class doesn't have a no-argument constructor? Recall that the no-argument constructor is not required and only inserted if there is no constructor defined in the class. In this case, the Java compiler will not help and you must create at least one constructor in your child class that explicitly calls a parent constructor via the super() command. For example, the following code will not compile:

```
public class Mammal {
public Mammal(int age) {
}
}
public class Elephant extends Mammal { // DOES NOT COMPILE
}
```

In this example no constructor is defined within the Elephant class, so the compiler tries to insert a default no-argument constructor with a super() call, as it did in the Donkey example. The compiler stops, though, when it realizes there is no parent constructor that takes no arguments. In this example, we must explicitly define at least one constructor, as in the following code:

```
public class Mammal {
public Mammal(int age) {
}
}
public class Elephant extends Mammal {
public Elephant() { // DOES NOT COMPILE
}
}
```

This code still doesn't compile, though, because the compiler tries to insert the noargument super() as the first statement of the constructor in the Elephant class, and there is no such constructor in the parent class. We can fix this, though, by adding a call to a parent constructor that takes a fixed argument:

```
public class Mammal {
public Mammal(int age) {
}
}
public class Elephant extends Mammal {
public Elephant() {
super(10);
}
}
```

This code will compile.

Constructor Definition Rules:

- The first statement of every constructor is a call to another constructor within the class using this(), or a call to a constructor in the direct parent class using super().
- The super() call may not be used after the first statement of the constructor.
- If no super() call is declared in a constructor, Java will insert a no-argument super() as the first statement of the constructor.
- If the parent doesn't have a no-argument constructor and the child doesn't define any constructors, the compiler will throw an error and try to insert a default no-argument constructor into the child class.
- If the parent doesn't have a no-argument constructor, the compiler requires an explicit call to a parent constructor in each child constructor.

```
class Primate {
public Primate() {
System.out.println("Primate");
}
}
class Ape extends Primate {
public Ape() {
System.out.println("Ape");
}
}
public class Chimpanzee extends Ape {
public static void main(String[] args) {
new Chimpanzee();
}
}
```

The compiler first inserts the super() command as the first statement of both the Primate and Ape constructors. Next, the compiler inserts a default no-argument constructor in the Chimpanzee class with super() as the first statement of the constructor. The code will execute with the parent constructors called first and yields the following output:

```
Primate
Ape
```

Java classes may use any public or protected member of the parent class, including methods, primitives, or object references. If the parent class and child class are part of the same package, the child class may also use any default members defined in the parent class. Finally, a child class may never access a private member of the parent class, at least not through any direct reference. To reference a member in a parent class, you can just call it directly.

You can use the keyword this to access a member of the class. You may also use this to access members of the parent class that are accessible from the child class, since a child class inherits all of its parent members. In Java, you can explicitly reference a member of the parent class by using the super keyword. We could use this or super to access a member of the parent class, but same is not true for a member of the child class.

In other words, we see that this and super may both be used for methods or variables defined in the parent class, but only this may be used for members defined in the current class. If the child class overrides a member of the parent class, this and super could have very different effects when applied to a class member.

this() and this are unrelated in Java. Likewise, super() and super are quite different but may be used in the same methods on the exam. The first, super(), is a statement that explicitly calls a parent constructor and may only be used in the first line of a constructor of a child class. The second, super, is a keyword used to reference a member defined in a parent class and may be used throughout the child class.

The compiler performs the following checks when you override a nonprivate method:

- ➢ The method in the child class must have the same signature as the method in the parent class.
- ➢ The method in the child class must be at least as accessible or more accessible than the method in the parent class.
- ➢ The method in the child class may not throw a checked exception that is new or broader than the class of any exception thrown in the parent class method.
- ➢ If the method returns a value, it must be the same or a subclass of the method in the parent class, known as *covariant return types*.

If two methods have the same name but different signatures, the methods are overloaded, not overridden. As you may recall from our discussion of overloaded methods in Chapter 4, the methods are unrelated to each other and do not share any properties.

Java permits you to redeclare a new method in the child class with the same or modified signature as the private method in the parent class. This method in the child class is a separate and independent method, unrelated to the parent version's method, so none of the rules for overriding methods are invoked.

A *hidden method* occurs when a child class defines a static method with the same name and signature as a static method defined in a parent class. Method hiding is similar but not exactly the same as method overriding. First, the four previous rules for overriding a method must be

followed when a method is hidden. In addition, a new rule is added for hiding a method, namely that the usage of the static keyword must be the same between
parent and child classes.

```
public class Bear {
public static void sneeze() {
System.out.println("Bear is sneezing");
}
public void hibernate() {
System.out.println("Bear is hibernating");
}
}
public class Panda extends Bear {
public void sneeze() { // DOES NOT COMPILE
System.out.println("Panda bear sneezes quietly");
}
public static void hibernate() { // DOES NOT COMPILE
System.out.println("Panda bear is going to sleep");
}
}
```

In our description of hiding of static methods, we indicated there was a distinction between overriding and hiding methods. Unlike overriding a method, in which a child method replaces the parent method in calls defined in both the parent and child, hidden methods only replace parent methods in the calls defined in the child class.

At runtime the child version of an overridden method is always executed for an instance regardless of whether the method call is defined in a parent or child class method. In this manner, the parent method is never used unless an explicit call to the parent method is referenced, using the syntax ParentClassName.method(). Alternatively, at runtime the parent version of a hidden method is always executed if the call to the method is defined in the parent class.

We conclude our discussion of method inheritance with a somewhat self-explanatory rule: final methods cannot be overridden. This rule is in place both when you override a method and when you hide a method. In other words, you cannot hide a static method in a parent class if it is marked as final.

Java doesn’t allow variables to be overridden but instead hidden. When you hide a variable, you define a variable with the same name as a variable in a parent class. This creates two copies of the variable within an instance of the child class: one instance defined for the parent reference and another defined for the child reference. As when hiding a static method, you can’t override a variable; you can only hide it. Also similar to hiding a static method, the rules for accessing the parent and child variables are quite similar. If you’re referencing the variable from within the parent class, the variable defined in the parent class is used. Alternatively, if you’re referencing the variable from within a child class, the variable defined in the child class is used. Likewise, you can reference the parent value of the variable with an explicit use of the super keyword. These rules are the same regardless of whether the variable is an instance variable or a static variable.

```
public class Animal {
public int length = 2;
}
public class Jellyfish extends Animal {
public int length = 5;
public static void main(String[] args) {
Jellyfish jellyfish = new Jellyfish();
Animal animal = new Jellyfish();
System.out.println(jellyfish.length);
System.out.println(animal.length);
}
}
```

This code compiles without issue. Here's the output:

```
5
2
```

Notice the same type of object was created twice, but the reference to the object determines which value is seen as output. If the object Jellyfish was passed to a method by an Animal reference, as you'll see in the section "Understanding Polymorphism," later in this chapter, the wrong value might be used.

abstract class is a class that is marked with the abstract keyword and cannot be instantiated. An *abstract method* is a method marked with the abstract keyword defined in an abstract class, for which no implementation is provided in the class in which it is declared.

```
public abstract class Animal {
protected int age;
public void eat() {
System.out.println("Animal is eating");
}
public abstract String getName();
}
public class Swan extends Animal {
public String getName() {
return "Swan";
}
}
```

The first thing to notice about this sample code is that the Animal class is declared abstract and Swan is not. Next, the member age and the method eat() are marked as protected and public, respectively; therefore, they are inherited in subclasses such as Swan. Finally, the abstract method getName() is terminated with a semicolon and doesn't provide a body in the parent class

Animal. This method is implemented with the same name and signature as the parent method in the Swan class.

An abstract class may include nonabstract methods and variables, as you saw with the variable age and the method eat(). In fact, an abstract class is not required to include any abstract methods. Although an abstract class doesn't have to implement any abstract methods, an abstract method may only be defined in an abstract class.

```
public class Chicken {
public abstract void peck(); // DOES NOT COMPILE
}

public abstract class Turtle {
public abstract void swim() {} // DOES NOT COMPILE
public abstract int getAge() { // DOES NOT COMPILE
return 10;
}
}
```

Next, we note that an abstract class cannot be marked as final for a somewhat obvious reason. By definition, an abstract class is one that must be extended by another class to be instantiated, whereas a final class can't be extended by another class. By marking an abstract class as final, you're saying the class can never be instantiated, so the compiler refuses to process the code.

```
public final abstract class Tortoise { // DOES NOT COMPILE
}
```

Likewise, an abstract method may not be marked as final for the same reason that an abstract class may not be marked as final. Once marked as final, the method can never be overridden in a subclass, making it impossible to create a concrete instance of the abstract class.

```
public abstract class Goat {
public abstract final void chew(); // DOES NOT COMPILE
}
```

Finally, a method may not be marked as both abstract and private.

```
public abstract class Whale {
private abstract void sing(); // DOES NOT COMPILE
}
```

Even with abstract methods, the rules for overriding methods must be followed.

When working with abstract classes, it is important to remember that by themselves, they cannot be instantiated and therefore do not do much other than define static variables and methods.

```
public abstract class Eel {
public static void main(String[] args) {
final Eel eel = new Eel(); // DOES NOT COMPILE
}
}
```

An abstract class becomes useful when it is extended by a concrete subclass. A *concrete class* is the first nonabstract subclass that extends an abstract class and is required to implement all inherited abstract methods.

```
public abstract class Animal {
public abstract String getName();
}
public class Walrus extends Animal { // DOES NOT COMPILE
}
```

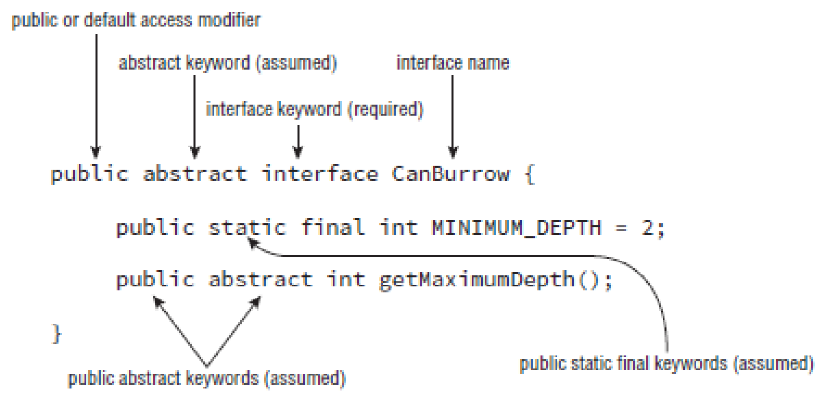
First, note that Animal is marked as abstract and Walrus is not. In this example, Walrus is considered the first concrete subclass of Animal. Second, since Walrus is the first concrete subclass, it must implement all inherited abstract methods, getName() in this example. Because it doesn't, the compiler rejects the code. Notice that when we define a concrete class as the "first" nonabstract subclass, we include the possibility that another nonabstract class may extend an existing nonabstract class. The key point is that the first class to extend the nonabstract class must implement all inherited abstract methods.

abstract classes can extend other abstract classes and are not required to provide implementations for any of the abstract methods. It follows, then, that a concrete class that extends an abstract class must implement all inherited abstract methods.

There is one exception to the rule for abstract methods and concrete classes: a concrete subclass is not required to provide an implementation for an abstract method if an intermediate abstract class provides the implementation.

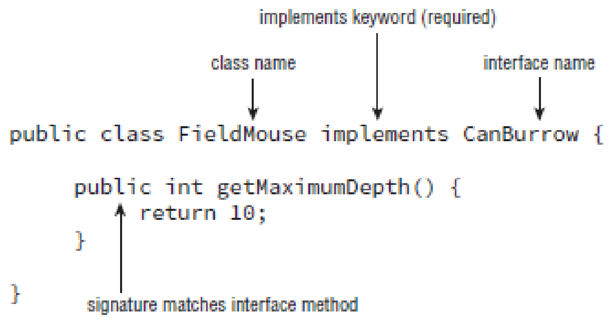
Abstract classes may not be marked as private or final.
Although Java doesn't allow multiple inheritance, it does allow classes to implement any number of interfaces. An *interface* is an abstract data type that defines a list of abstract public methods that any class implementing the interface must provide. An interface can also include a list of constant variables and default methods. In Java, an interface is defined with the interface keyword, analogous to the class keyword used when defining a class. A class invokes the interface by using the implements keyword in its class definition.

FIGURE 5.4 Defining an interface


public or default access modifier
abstract keyword (assumed)
interface name
interface keyword (required)
public static final keywords (assumed)
public abstract keywords (assumed)

```
public abstract interface CanBurrow {
    public static final int MINIMUM_DEPTH = 2;
    public abstract int getMaximumDepth();
}
```

FIGURE 5.5 Implementing an interface


implements keyword (required)
class name
interface name
signature matches interface method

```
public class FieldMouse implements CanBurrow {

     public int getMaximumDepth() {
          return 10;
     }

}
```

Notice that the method modifiers in this example, abstract and public, are assumed. In other words, whether or not you provide them, the compiler will automatically insert them as part of the method definition. A class may implement multiple interfaces, each separated by a comma…
public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {
}

List of rules for creating an interface…

1. Interfaces cannot be instantiated directly.
2. An interface is not required to have any methods.
3. An interface may not be marked as final.
4. All top-level interfaces are assumed to have public or default access, and they must include the abstract modifier in their definition. Therefore, marking an interface as private, protected, or final will trigger a compiler error, since this is incompatible with these assumptions.
5. All nondefault methods in an interface are assumed to have the modifiers abstract and public in their definition. Therefore, marking a method as private, protected, or final will trigger compiler errors as these are incompatible with the abstract and public keywords.

The fourth rule doesn't apply to inner interfaces, although inner classes and interfaces are not in scope for the OCA exam. The first three rules are identical to the first three rules for creating an abstract class.

Following two interface definitions are equivalent, as the compiler will convert them both to the second example:
public interface CanFly {
void fly(int speed);
abstract void takeoff();
public abstract double dive();
}
public abstract interface CanFly {
public abstract void fly(int speed);
public abstract void takeoff();

```
public abstract double dive();
}

private final interface CanCrawl { // DOES NOT COMPILE
private void dig(int depth); // DOES NOT COMPILE
protected abstract double depth(); // DOES NOT COMPILE
public final void surface(); // DOES NOT COMPILE
}
```

There are two inheritance rules you should keep in mind when extending an interface:

1. An interface that extends another interface, as well as an abstract class that implements an interface, inherits all of the abstract methods as its own abstract methods.
2. The first concrete class that implements an interface, or extends an abstract class that implements an interface, must provide an implementation for all of the inherited abstract methods.

Like an abstract class, an interface may be extended using the extend keyword. In this manner, the new child interface inherits all the abstract methods of the parent interface. Unlike an abstract class, though, an interface may extend multiple interfaces. Consider the following example:

```
public interface HasTail {
public int getTailLength();
}
public interface HasWhiskers {
public int getNumberOfWhiskers();
}
public interface Seal extends HasTail, HasWhiskers {
}
```

Any class that implements the Seal interface must provide an implementation for all methods in the parent interfaces—in this case, getTailLength() and getNumberOfWhiskers(). What about an abstract class that implements an interface? In this scenario, the abstract class is treated in the same way as an interface extending another interface. In other words, the abstract class inherits the abstract methods of the interface but is not required to implement them. That said, like an abstract class, the first concrete class to extend the abstract class must implement all the inherited abstract methods of the interface.

Although a class can implement an interface, a class cannot extend an interface. Likewise, whereas an interface can extend another interface, an interface cannot implement another interface.

Only connection between a class and an interface is with the *class* implements *interface* syntax.

If two abstract interface methods have identical behaviors—or in this case the same method signature— creating a class that implements one of the two methods automatically implements the second method. If the method name is the same but the input parameters are different, there is no conflict because this is considered a method overload.

Notice that for the overloaded methods, it doesn't matter if the return type of the two methods is the same or different, because the compiler treats these methods as independent. Unfortunately, if the method name and input parameters are the same but the return types are different between the two methods, the class or interface attempting to inherit both interfaces will not compile. It is not possible in Java to define two methods in a class with the same name and input parameters but different return types.

```
public interface Herbivore {
public int eatPlants();
}
public interface Omnivore {
public void eatPlants();
}
public class Bear implements Herbivore, Omnivore {
public int eatPlants() { // DOES NOT COMPILE
System.out.println("Eating plants: 10");
return 10;
}
public void eatPlants() { // DOES NOT COMPILE
System.out.println("Eating plants");
}
}
```

If we were to remove either definition of eatPlants(), the compiler would stop because the definition of Bear would be missing one of the required methods. In other words, there is no implementation of the Bear class that inherits from Herbivore and Omnivore that the compiler would accept. The compiler would also throw an exception if you define an interface or abstract class that inherits from two conflicting interfaces.

```
public interface Herbivore {
public int eatPlants();
}
public interface Omnivore {
public void eatPlants();
}
public interface Supervore extends Herbivore, Omnivore {} // DOES NOT COMPILE
public abstract class AbstractBear implements Herbivore, Omnivore {} // DOES NOT COMPILE
```

Like interface methods, interface variables are assumed to be public. Unlike interface methods, though, interface variables are also assumed to be static and final.
Here are two interface variables rules:

1. Interface variables are assumed to be public, static, and final. Therefore, marking a variable as private or protected will trigger a compiler error, as will marking any variable as abstract.
2. The value of an interface variable must be set when it is declared since it is marked as final.

The compiler will automatically insert public static final to any constant interface variables it finds missing those modifiers.

```
public interface CanDig {
private int MAXIMUM_DEPTH = 100; // DOES NOT COMPILE
protected abstract boolean UNDERWATER = false; // DOES NOT COMPILE
public static String TYPE; // DOES NOT COMPILE
}
```

The second line, UNDERWATER, doesn't compile for two reasons. It is marked as protected, which conflicts with the assumed modifier public, and it is marked as abstract, which conflicts with the assumed modifier final.

A *default method* is a method defined within an interface with the default keyword in which a method body is provided. Contrast default methods with "regular" methods in an interface, which are assumed to be abstract and may not have a method body.

A default method within an interface defines an abstract method with a default implementation. In this manner, classes have the option to override the default method if they need to, but they are not required to do so. If the class doesn't override the method, the default implementation will be used. In this manner, the method definition is concrete, not abstract.

The purpose of adding default methods to the Java language was in part to help with code development and backward compatibility. Imagine you have an interface that is shared among dozens or even hundreds of users that you would like to add a new method to. If you just update the interface with the new method, the implementation would break among all of your subscribers, who would then be forced to update their code. In practice, this might even discourage you from making the change altogether. By providing a default implementation of the method, though, the interface becomes backward compatible with the existing codebase, while still providing those individuals who do want to use the new method with the option to override it.

The following are the default interface method rules you need to be familiar with:

1. A default method may only be declared within an interface and not within a class or abstract class.
2. A default method must be marked with the default keyword. If a method is marked as default, it must provide a method body.
3. A default method is not assumed to be static, final, or abstract, as it may be used or overridden by a class that implements the interface.

4. Like all methods in an interface, a default method is assumed to be public and will not compile if marked as private or protected.

```
public interface Carnivore {
public default void eatMeat(); // DOES NOT COMPILE
public int getRequiredFoodAmount() { // DOES NOT COMPILE
return 13;
}
}
```

Unlike interface variables, which are assumed static class members, default methods cannot be marked as static and require an instance of the class implementing the interface to be invoked. They can also not be marked as final or abstract, because they are allowed to be overridden in subclasses but are not required to be overridden.

When an interface extends another interface that contains a default method, it may choose to ignore the default method, in which case the default implementation for the method will be used. Alternatively, the interface may override the definition of the default method using the standard rules for method overriding, such as not limiting the accessibility of the method and using covariant returns. Finally, the interface may redeclare the method as abstract, requiring classes that implement the new interface to explicitly provide a method body.

```
public interface HasFins {
public default int getNumberOfFins() {
return 4;
}
public default double getLongestFinLength() {
return 20.0;
}
public default boolean doFinsHaveScales() {
return true;
}
}
public interface SharkFamily extends HasFins {
public default int getNumberOfFins() {
return 8;
}
public double getLongestFinLength();
public boolean doFinsHaveScales() { // DOES NOT COMPILE
return false;
}
}
```

In this example, the first interface, HasFins, defines three default methods: getNumberOfFins(), getLongestFinLength(), and doFinsHaveScales(). The second interface, SharkFamily, extends HasFins and overrides the default method getNumberOfFins() with a new method that returns a different value. Next, the SharkFamily interface replaces the default method getLongestFinLength() with a new abstract method, forcing any class that implements the SharkFamily interface to provide an implementation of the method. Finally, the SharkFamily interface overrides the doFinsHaveScales() method but doesn't mark the method as default. Since interfaces may only contain methods with a body that are marked as default, the code will not compile.

If a class implements two interfaces that have default methods with the same name and signature, the compiler will throw an error. There is an exception to this rule, though: if the subclass overrides the duplicate default methods, the code will compile without issue—the ambiguity about which version of the method to call has been removed.

A class that implements or inherits two duplicate default methods forces the class to implement a new version of the method, or the code will not compile. This rule holds true even for abstract classes that implement multiple interfaces, because the default method could be called in a concrete method within the abstract class.

Java 8 also now includes support for static methods within interfaces. In fact, there is really only one distinction between a static method in a class and an interface. A static method defined in an interface is not inherited in any classes that implement the interface.

1. Like all methods in an interface, a static method is assumed to be public and will not compile if marked as private or protected.
2. To reference the static method, a reference to the name of the interface must be used.

A class that implements two interfaces containing static methods with the same signature will still compile at runtime, because the static methods are not inherited by the subclass and must be accessed with a reference to the interface name.

Java supports *polymorphism*, the property of an object to take on many different forms. To put this more precisely, a Java object may be accessed using a reference with the same type as the object, a reference that is a superclass of the object, or a reference that defines an interface the object implements, either directly or through a superclass. Furthermore, a cast is not required if the object is being reassigned to a super type or interface of the object.

```
public class Primate {
public boolean hasHair() {
return true;
}
}
public interface HasTail {
public boolean isTailStriped();
}
public class Lemur extends Primate implements HasTail {
public boolean isTailStriped() {
```

```
return false;
}
public int age = 10;
public static void main(String[] args) {
Lemur lemur = new Lemur();
System.out.println(lemur.age);
HasTail hasTail = lemur;
System.out.println(hasTail.isTailStriped());
Primate primate = lemur;
System.out.println(primate.hasHair());
}
}
```

This code compiles and executes without issue and yields the following output:

```
10
false
true
```

The most important thing to note about this example is that only one object, Lemur, is created and referenced. The ability of an instance of Lemur to be passed as an instance of an interface it implements, HasTail, as well as an instance of one of its superclasses, Primate, is the nature of polymorphism.

```
HasTail hasTail = lemur;
System.out.println(hasTail.age); // DOES NOT COMPILE
Primate primate = lemur;
System.out.println(primate.isTailStriped()); // DOES NOT COMPILE
```

In Java, all objects are accessed by reference, so as a developer you never have direct access to the object itself. Conceptually, though, you should consider the object as the entity that exists in memory, allocated by the Java runtime environment. Regardless of the type of the reference you have for the object in memory, the object itself doesn't change. For example, since all objects inherit java.lang.Object, they can all be reassigned to java.lang.Object, as shown in the following example:

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;
```

We can summarize this principle with the following two rules:

1. The type of the object determines which properties exist within the object in memory.
2. The type of the reference to the object determines which methods and variables are accessible to the Java program.

It therefore follows that successfully changing a reference of an object to a new reference type may give you access to new properties of the object, but those properties existed before the reference change occurred.

Once we changed the reference type, though, we lost access to more specific methods defined in the subclass that still exist within the object. We can reclaim those references by casting the object back to the specific subclass it came from:

```
Primate primate = lemur;
Lemur lemur2 = primate; // DOES NOT COMPILE
Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

Here are some basic rules to keep in mind when casting variables:

- Casting an object from a subclass to a superclass doesn't require an explicit cast.
- Casting an object from a superclass to a subclass requires an explicit cast.
- The compiler will not allow casts to unrelated types.
- Even when the code compiles without issue, an exception may be thrown at runtime if the object being cast is not actually an instance of that class.

```
public class Rodent {
}
public class Capybara extends Rodent {
public static void main(String[] args) {
Rodent rodent = new Rodent();
Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime
}
}
```

This code creates an instance of Rodent and then tries to cast it to a subclass of Rodent, Capybara. Although this code will compile without issue, it will throw a ClassCastException at runtime since the object being referenced is not an instance of the Capybara class. The thing to keep in mind in this example is the object that was created is not related to the Capybara class in any way.

instanceof operator can be used to check whether an object belongs to a particular class and to prevent ClassCastExceptions at runtime. Unlike the previous example, the following code snippet doesn't throw an exception at runtime and performs the cast only if the instanceof operator returns true.

```
if(rodent instanceof Capybara) {
Capybara capybara = (Capybara)rodent;
}
```

The most important feature of polymorphism—and one of the primary reasons we have class structure at all—is to support virtual methods. A *virtual method* is a method in which the specific implementation is not determined until runtime. In fact, all non-final, nonstatic, and non-private Java methods are considered virtual methods, since any of them can be overridden at runtime. What makes a virtual method special in Java is that if you call a method on an object that overrides a method, you get the overridden method, even if the call to the method is on a parent reference or within the parent class.

```
public class Bird {
public String getName() {
return "Unknown";
}
public void displayInformation() {
System.out.println("The bird name is: "+getName());
}
}
public class Peacock extends Bird {
public String getName() {
return "Peacock";
}
public static void main(String[] args) {
Bird bird = new Peacock();
bird.displayInformation();
}
}
```

This code compiles and executes without issue and outputs the following:

```
The bird name is: Peacock
```

One of the most useful applications of polymorphism is the ability to pass instances of a subclass or interface to a method. For example, you can define a method that takes an instance of an interface as a parameter. In this manner, any class that implements the interface can be passed to the method. Since you're casting from a subtype to a supertype, an explicit cast is not required. This property is referred to as *polymorphic parameters* of a method

```
public class Reptile {
public String getName() {
return "Reptile";
}
}
public class Alligator extends Reptile {
public String getName() {
return "Alligator";
}
}
public class Crocodile extends Reptile {
public String getName() {
return "Crocodile";
}
}
public class ZooWorker {
public static void feed(Reptile reptile) {
System.out.println("Feeding reptile "+reptile.getName());
}
public static void main(String[] args) {
feed(new Alligator());
feed(new Crocodile());
feed(new Reptile());
}
}
```

This code compiles and executes without issue, yielding the following output:

```
Feeding: Alligator
Feeding: Crocodile
Feeding: Reptile
```

If the return type of a method is Double in the parent class and is overridden in a subclass with a method that returns Number, a superclass of Double, then the subclass method would be allowed to return any valid Number, including Integer, another subclass of Number. If we are using the object with a reference to the superclass, that means an Integer could be returned when a Double was expected. Since Integer is not a subclass of Double, this would lead to an implicit cast exception as soon as the value was referenced. Java solves this problem by only allowing covariant return types for overridden methods.

Exceptions

An exception indicates something unexpected happened. A method can handle an exception by catching it or declaring it for the caller to deal with. Many exceptions are thrown by Java libraries. You can throw your own exception with code such as throw new Exception().

Subclasses of java.lang.Error are exceptions that a programmer should not attempt to handle. Subclasses of java.lang.RuntimeException are runtime (unchecked) exceptions. Subclasses of java.lang.Exception, but not java.lang.RuntimeException are checked exceptions. Java requires checked exceptions to be handled or declared.

If a try statement has multiple catch blocks, at most one catch block can run. Java looks for an exception that can be caught by each catch block in the order they appear, and the first match is run. Then execution continues after the try statement. If both catch and finally throw an exception, the one from finally gets thrown.

Common runtime exceptions include:

- ArithmeticException
- ArrayIndexOutOfBoundsException
- ClassCastException
- IllegalArgumentException
- NullPointerException
- NumberFormatException

IllegalArgumentException and NumberFormatException are typically thrown by the programmer, whereas the others are typically thrown by the JVM.

Common checked exceptions include:

- IOException
- FileNotFoundException

Common errors include:

- ExceptionInInitializerError
- StackOverflowError
- NoClassDefFoundError

When a method overrides a method in a superclass or interface, it is not allowed to add checked exceptions. It is allowed to declare fewer exceptions or declare a subclass of a declared exception. Methods declare exceptions with the keyword throws.

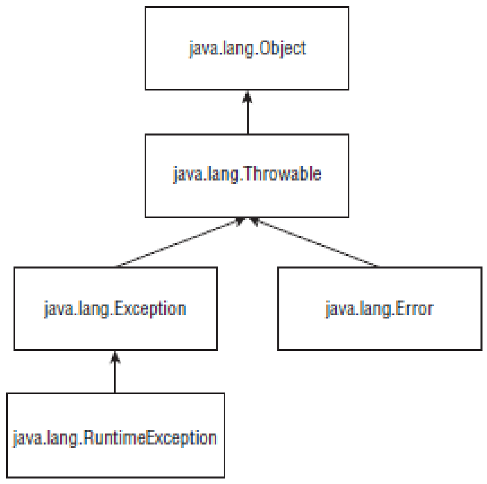
A try statement must have a catch or a finally block. Multiple catch blocks are also allowed, provided no superclass exception type appears in an earlier catch block than its subclass. The finally block runs last regardless of whether an exception is thrown.

The throw keyword is used when you actually want to throw an exception—for example**,** throw new RuntimeException()**.** The throws keyword is used in a method declaration.

An *exception* is Java's way of saying, "I give up. I don't know what to do right now. You deal with it." When you write a method, you can either deal with the exception or make it the calling code's problem.

An exception is an event that alters program flow. Java has a Throwable superclass for all objects that represent these events. Not all of them have the word exception in their classname, which can be confusing.

FIGURE 6.1 Categories of exception



Error means something went so horribly wrong that your program should not attempt to recover from it.

A *runtime exception* is defined as the RuntimeException class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Runtime exceptions are also known as *unchecked exceptions*.

A runtime (unchecked) exception is a specific type of exception. All exceptions occur at the time that the program is run. (The alternative is compile time, which would be a compiler error.) People don't refer to them as run time exceptions because that would be too easy to confuse with runtime! When you see runtime, it means unchecked.

A *checked exception* includes Exception and all subclasses that do not extend RuntimeException. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn't exist. Checked exceptions? What are we checking? Java has a rule called the *handle or declare rule*. For checked exceptions, Java requires the code to either handle them or declare them in the method signature. For example, this method declares that it might throw an exception:

```
void fall() throws Exception {
throw new Exception();
}
```

Notice that you're using two different keywords here. throw tells Java that you want to throw an Exception. throws simply declares that the method might throw an Exception. It also might not. Because checked exceptions tend to be anticipated, Java enforces that the programmer do something to show the exception was thought about. Maybe it was handled in the method. Or maybe the method declares that it can't handle the exception and someone else should.

An example of a runtime exception is a NullPointerException, which happens when you try to call a member on a null reference. This can occur in any method. If you had to declare runtime exceptions everywhere, every single method would have that clutter!

On the exam, you will see two types of code that result in an exception. The first is code that's wrong. The second way for code to result in an exception is to explicitly request Java to throw one. Java lets you write statements like these:

```
throw new Exception();
throw new Exception("Ow! I fell.");
throw new RuntimeException();
throw new RuntimeException("Ow! I fell.");
```

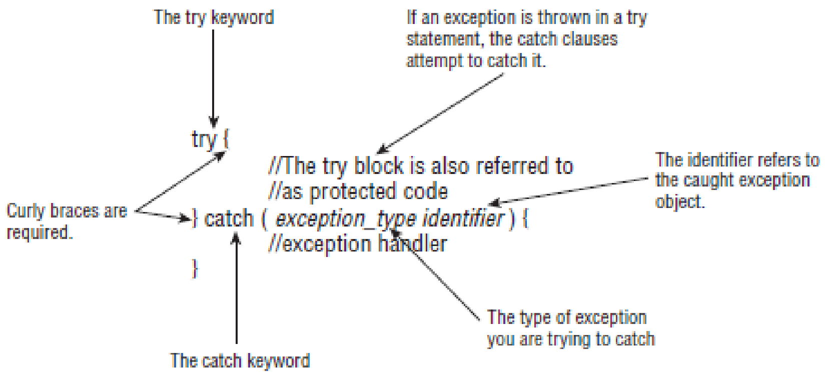
The throw keyword tells Java you want some other part of the code to deal with the exception. When creating an exception, you can usually pass a String parameter with a message or you can pass no parameters and use the defaults.

TABLE 6.1 Types of exceptions

Type	How to recognize	Okay for program to catch?	Is program required to handle or declare?
Runtime exception	Subclass of `RuntimeException`	Yes	No
Checked exception	Subclass of `Exception` but not subclass of `RuntimeException`	Yes	Yes
Error	Subclass of `Error`	No	No

Java uses a *try statement* to separate the logic that might throw an exception from the logic to handle that exception.

FIGURE 6.2 The syntax of a *try* statement


The try keyword
If an exception is thrown in a try statement, the catch clauses attempt to catch it.
try {
//The try block is also referred to
//as protected code
The identifier refers to the caught exception object.
Curly braces are required.
} catch (exception_type identifier) {
//exception handler
}
The type of exception you are trying to catch
The catch keyword

The code in the try block is run normally. If any of the statements throw an exception that can be caught by the exception type listed in the catch block, the try block stops running and execution goes to the catch statement. If none of the statements in the try block throw an exception that can be caught, the *catch clause* is not run.

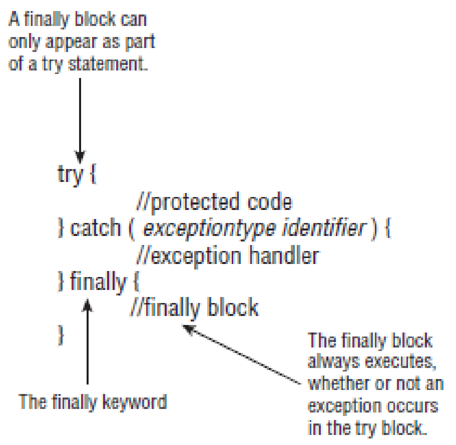
You probably noticed the words "block" and "clause" used interchangeably. The exam does this as well, so we are getting you used to it. Both are correct. "Block" is correct because there are braces present. "Clause" is correct because they are part of a try statement.

```
try // DOES NOT COMPILE
fall();
catch (Exception e)
System.out.println("get up");
```
The problem is that the braces are missing for both try and catch clauses.

try statements are like methods in that the curly braces are required even if there is only one statement inside the code blocks. if statements and loops are special in this respect as they allow you to omit the curly braces.
```
try {// DOES NOT COMPILE
fall();
}
```
This code doesn't compile because the try block doesn't have anything after it. Remember, the point of a try statement is for something to happen if an exception is thrown. Without another clause, the try statement is lonely.

The try statement also lets you run code at the end with a *finally clause* regardless of whether an exception is thrown.

FIGURE 6.3 The syntax of a *try* statement with *finally*



```
try {
        //protected code
} catch ( exceptiontype identifier ) {
        //exception handler
} finally {
        //finally block
}
```

A finally block can only appear as part of a try statement.
The finally keyword
The finally block always executes, whether or not an exception occurs in the try block.

There are two paths through code with both a catch and a finally. If an exception is thrown, the finally block is run after the catch block. If no exception is thrown, the finally block is run after the try block completes.

On the OCA exam, a try statement must have catch and/or finally. Having both is fine. Having neither is a problem. On the OCP exam, you'll learn about a special syntax for a try statement called try-with-resources that allows neither a catch nor a finally block. On the OCA exam, you get to assume a try statement is just a regular try statement and not a try-with-resources statement.

```
25: try { // DOES NOT COMPILE
26: fall();
27: } finally {
28: System.out.println("all better");
29: } catch (Exception e) {
30: System.out.println("get up");
31: }
32:
33: try { // DOES NOT COMPILE
34: fall();
35: }
36:
37: try {
38: fall();
39: } finally {
40: System.out.println("all better");
41: }
```

The first example (lines 25–31) does not compile because the catch and finally blocks are in the wrong order. The second example (lines 33–35) does not compile because there must be a catch or finally block. The third example (lines 37–41) is just fi ne. catch is not required if finally is present.

One problem with finally is that any realistic uses for it are out of the scope of the OCA exam. finally is typically used to close resources such as fi les or databases—both of which are topics on the OCP exam. This means most of the examples you encounter on the OCA exam with finally are going to look contrived.

There is one exception to “the finally block always runs after the catch block” rule: Java defines a method that you call as System.exit(0);. The integer parameter is the error code that gets returned. System.exit tells Java, “Stop. End the program right now. Do not pass go. Do not collect $200.” When System.exit is called in the try or catch block, finally does not run.

A rule exists for the order of the catch blocks. Java looks at them in the order they appear. If it is impossible for one of the catch blocks to be executed, a compiler error about unreachable code occurs. This happens when a superclass is caught before a subclass.

A catch or finally block can have any valid Java code in it—including another try statement.

```
26: try {
27: throw new RuntimeException();
28: } catch (RuntimeException e) {
29: throw new RuntimeException();
30: } finally {
31: throw new Exception();
32: }
```

Line 27 throws an exception, which is caught on line 28. The catch block then throws an exception on line 29. If there were no finally block, the exception from line 29 would be thrown. However, the finally block runs after the try block. Since the finally block throws an exception of its own on line 31, this one gets thrown. The exception from the catch block gets forgotten about. This is why you often see another try/catch inside a finally block—to make sure it doesn’t mask the exception from the catch block.

Runtime exceptions don’t have to be handled or declared. They can be thrown by the programmer or by the JVM.

ArithmeticException Thrown by the JVM when code attempts to divide by zero.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

The thread "main" is telling us the code was called directly or indirectly from a program with a main method.

ArrayIndexOutOfBoundsException Thrown by the JVM when code uses an illegal index to access an array

ClassCastException Thrown by the JVM when an attempt is made to cast an object to a subclass of which it is not an instance.

Java tries to protect you from impossible casts. This code doesn't compile because Integer is not a subclass of String:

```
String type = "moose";
Integer number = (Integer) type; // DOES NOT COMPILE
```

More complicated code thwarts Java's attempts to protect you. When the cast fails at runtime, Java will throw a ClassCastException:

```
String type = "moose";
Object obj = type;
Integer number = (Integer) obj;
```

The compiler sees a cast from Object to Integer. This could be okay. The compiler doesn't realize there's a String in that Object. When the code runs, it yields the following output:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to
java.lang.Integer
```

Java tells us both types that were involved in the problem, making it apparent what's wrong.

IllegalArgumentException Thrown by the programmer to indicate that a method has been passed an illegal or inappropriate argument

IllegalArgumentException is a way for your program to protect itself. We first saw the following setter method in the Swan class in Chapter 4, "Methods and Encapsulation."

```
6: public void setNumberEggs(int numberEggs) {// setter
7: if (numberEggs >= 0) // guard condition
8: this.numberEggs = numberEggs;
9: }
```

This code works, but we don't really want to ignore the caller's request when they tell us a Swan has –2 eggs. We want to tell the caller that something is wrong—preferably in a very obvious way that the caller can't ignore so that the programmer will fix the problem. Exceptions are an efficient way to do this. Seeing the code end with an exception is a great reminder that something is wrong:

```
public static void setNumberEggs(int numberEggs) {
if (numberEggs < 0)
throw new IllegalArgumentException(
"# eggs must not be negative");
this.numberEggs = numberEggs;
}
```

The program throws an exception when it's not happy with the parameter values. The output looks like this:

```
Exception in thread "main" java.lang.IllegalArgumentException: # eggs must not be negative
```

Clearly this is a problem that must be fixed if the programmer wants the program to do anything useful.

NullPointerException Thrown by the JVM when there is a null reference where an object is required.

```
String name;
public void printLength() throws NullPointerException {
System.out.println(name.length());
}
```

Running this code results in this output:

```
Exception in thread "main" java.lang.NullPointerException
```

NumberFormatException Thrown by the programmer when an attempt is made to convert a string to a numeric type but the string doesn't have an appropriate format

Java provides methods to convert strings to numbers. When these are passed an invalid value, they throw a NumberFormatException. The idea is similar to IllegalArgumentException. Since this is a common problem, Java gives it a separate class. In fact, NumberFormatException is a subclass of IllegalArgumentException. Here's an example of trying to convert something non-numeric into an int:

```
Integer.parseInt("abc");
```

The output looks like this:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
```

Checked exceptions have Exception in their hierarchy but not RuntimeException. They must be handled or declared. They can be thrown by the programmer or by the JVM.
FileNotFoundException Thrown programmatically when code tries to reference a file that does not exist
IOException Thrown programmatically when there's a problem reading or writing a file. For the OCA exam, you only need to know that these are checked exceptions. Also keep in mind that FileNotFoundException is a subclass of IOException

Errors
Errors extend the Error class. They are thrown by the JVM and should not be handled or declared. Errors are rare, but you might see these:

ExceptionInInitializerError Thrown by the JVM when a static initializer throws an exception and doesn't handle it

StackOverflowError Thrown by the JVM when a method calls itself too many times (this is called *infinite recursion* because the method typically calls itself without end)

NoClassDefFoundError Thrown by the JVM when a class that the code uses is available at compile time but not runtime

ExceptionInInitializerError

Java runs static initializers the first time a class is used. If one of the static initializers throws an exception, Java can't start using the class. It declares defeat by throwing an ExceptionInInitializerError. This code shows an ArrayIndexOutOfBounds in a static initializer:

```
static {
int[] countsOfMoose = new int[3];
int num = countsOfMoose[-1];
}

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

This code yields information about two exceptions:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: -1
```

We get the ExceptionInInitializerError because the error happened in a static initializer. That information alone wouldn't be particularly useful in fixing the problem. Therefore, Java also tells us the original cause of the problem: the ArrayIndexOutOfBoundsException that we need to fix.

The ExceptionInInitializerError is an error because Java failed to load the whole class. This failure prevents Java from continuing.

StackOverflowError

When Java calls methods, it puts parameters and local variables on the stack. After doing this a very large number of times, the stack runs out of room and overflows. This is called a StackOverflowError. Most of the time, this error occurs when a method calls itself.

```
public static void doNotCodeThis(int num) {
doNotCodeThis(1);
}
```

The output contains this line:

```
Exception in thread "main" java.lang.StackOverflowError
```

Since the method calls itself, it will never end. Eventually, Java runs out of room on the stack and throws the error. This is called infinite recursion. It is better than an infinite loop because at least Java will catch it and throw the error. With an infinite loop, Java just uses all your CPU until you can kill it.

NoClassDefFoundError

This error won't show up in code on the exam—you just need to know that it is an error. NoClassDefFoundError occurs when Java can't find the class at runtime.

```
class NoMoreCarrotsException extends Exception {}
public class Bunny {
public static void main(String[] args) {
eatCarrot();// DOES NOT COMPILE
}
private static void eatCarrot() throws NoMoreCarrotsException {
}
}
```

The problem is that NoMoreCarrotsException is a checked exception. Checked exceptions must be handled or declared. The code would compile if we changed the main() method to either of these:

```
public static void main(String[] args) throws NoMoreCarrotsException {// declare exception
eatCarrot();
}
public static void main(String[] args) {
try {
eatCarrot();
} catch (NoMoreCarrotsException e ) {// handle exception
System.out.print("sad rabbit");
}
}
```

You might have noticed that eatCarrot() didn't actually throw an exception; it just declared that it could. This is enough for the compiler to require the caller to handle or declare the exception. The compiler is still on the lookout for unreachable code. Declaring an unused exception isn't considered unreachable code. It gives the method the option to change the implementation to throw that exception in the future.

```
public void bad() {
try {
eatCarrot();
} catch (NoMoreCarrotsException e ) {// DOES NOT COMPILE
System.out.print("sad rabbit");
}
}
public void good() throws NoMoreCarrotsException {
eatCarrot();
}
private static void eatCarrot() { }
```

Java knows that eatCarrot() can't throw a checked exception—which means there's no way for the catch block in bad() to be reached. In comparison, good() is free to declare other exceptions.

When a class overrides a method from a superclass or implements a method from an interface, it's not allowed to add new checked exceptions to the method signature.

```
class CanNotHopException extends Exception { }
class Hopper {
public void hop() { }
}
class Bunny extends Hopper {
public void hop() throws CanNotHopException { } // DOES NOT COMPILE
}
```

A subclass is allowed to declare fewer exceptions than the superclass or interface. This is legal because callers are already handling them. This rule applies only to checked exceptions. The following code is legal because it has a runtime exception in the subclass's version:

```
class Hopper {
public void hop() { }
}
class Bunny extends Hopper {
public void hop() throws IllegalStateException { }
}
```

The reason that it's okay to declare new runtime exceptions in a subclass method is that the declaration is redundant. Methods are free to throw any runtime exceptions they want without mentioning them in the method declaration.

There are three ways to print an exception. You can let Java print it out, print just the message, or print where the stack trace comes from. This example shows all three approaches:

```
5: public static void main(String[] args) {
6: try {
7: hop();
8: } catch (Exception e) {
9: System.out.println(e);
10: System.out.println(e.getMessage());
11: e.printStackTrace();
12: }
13: }
14: private static void hop() {
15: throw new RuntimeException("cannot hop");
16: }
```

This code results in the following output:

```
java.lang.RuntimeException: cannot hop
cannot hop
java.lang.RuntimeException: cannot hop
at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

The first line shows what Java prints out by default: the exception type and message. The second line shows just the message. The rest shows a stack trace. The stack trace is usually the most helpful one because it shows where the exception occurred in each method that it passed through.

Every time you call a method, Java adds it to the stack until it completes. When an exception is thrown, it goes through the stack until it finds a method that can handle it or it runs out of stack.

Miscellaneous