

Chapter 3

Making Decisions

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Controlling Program Flow**

- Create program flow control constructs including if/else, switch statements and expressions, loops, and break and continue statements.

✓ **Using Object-Oriented Concepts in Java**

- Implement inheritance, including abstract and sealed types as well as record classes. Override methods, including that of the Object class. Implement polymorphism and differentiate between object type and reference type. Perform reference type casting, identify object types using the instanceof operator, and pattern matching with the instanceof operator and the switch construct.

Like many programming languages, Java is composed primarily of variables, operators, and statements put together in some logical order. In the previous chapter, we covered how to create and manipulate variables. Writing software is about more than managing variables, though; it is about creating applications that can make intelligent decisions. In this chapter, we present the various decision-making statements available to you within the language. This knowledge will allow you to build complex functions and class structures that you'll see throughout this book.

Creating Decision-Making Statements

Java operators allow you to create a lot of complex expressions, but they're limited in the manner in which they can control program flow. Imagine you

want a method to be executed only under certain conditions that cannot be evaluated until runtime. For example, on rainy days, a zoo should remind patrons to bring an umbrella, or on a snowy day, the zoo might need to close. The software doesn't change, but the behavior of the software should, depending on the inputs supplied in the moment. In this section, we discuss decision-making statements including `if` and `else`, along with pattern matching.

Statements and Blocks

As you may recall from [Chapter 1](#), “Building Blocks,” a Java statement is a complete unit of execution in Java, terminated with a semicolon (`;`). In this chapter, we introduce you to various Java control flow statements. *Control flow statements* break up the flow of execution by using decision-making, looping, and branching, allowing the application to selectively execute particular segments of code.

These statements can be applied to single expressions as well as a block of Java code. As described in [Chapter 1](#), a block of code in Java is a group of zero or more statements between balanced braces (`{}`) and can be used anywhere a single statement is allowed. For example, the following two snippets are equivalent, with the first being a single statement and the second being a block containing the same statement:

```
// Single statement
patrons++;

// Statement inside a block
{
    patrons++;
}
```

A statement or block often serves as the target of a decision-making statement. For example, we can prepend the decision-making `if` statement to these two examples:

```
// Single statement
if (ticketsTaken > 1)
    patrons++;

// Statement inside a block
if (ticketsTaken > 1) {
```

```
    patrons++;  
}
```

Again, both of these code snippets are equivalent. Just remember that the target of a decision-making statement can be a single statement or block of statements. For the rest of the chapter, we use both forms to better prepare you for what you will see on the exam.



While both of the previous examples are equivalent, stylistically using blocks is often preferred, even if the block has only one statement. The second form has the advantage that you can quickly insert new lines of code into the block, without modifying the surrounding structure.

The *if* Statement

Often, we want to execute a block only under certain circumstances. The `if` statement, as shown in [Figure 3.1](#), accomplishes this by allowing our application to execute a particular block of code if and only if a boolean expression evaluates to `true` at runtime.



A sample structure of an `if` statement. It reads `if` as `if` keyword, boolean expression as parenthesis, and braces as a block of multiple statements of optional for single statement.

FIGURE 3.1 The structure of an `if` statement

For example, imagine we had a function that used the hour of day, an integer value from 0 to 23, to display a message to the user:

```
if (hourOfDay < 11)  
    System.out.println("Good Morning");
```

If the hour of the day is less than 11, then the message will be displayed. Now let's say we also wanted to increment some value, `morningGreetingCount`, every time the greeting is printed. We could write

the `if` statement twice, but luckily Java offers us a more natural approach using a block:

```
if (hourOfDay < 11) {  
    System.out.println("Good Morning");  
    morningGreetingCount++;  
}
```

Watch Indentation and Braces

One area where the exam writers will try to trip you up is `if` statements without braces (`{}`). For example, take a look at this slightly modified form of our example:

```
if (hourOfDay < 11)  
    System.out.println("Good Morning");  
    morningGreetingCount++;
```

Based on the indentation, you might be inclined to think the variable `morningGreetingCount` is only going to be incremented if `hourOfDay` is less than 11, but that's not what this code does. It will execute the print statement only if the condition is met, but it will *always* execute the increment operation.

Remember that in Java, unlike some other programming languages, tabs are just whitespace and are not evaluated as part of the execution. When you see a control flow statement in a question, be sure to trace the open and close braces of the block, ignoring any indentation you may come across.


The `else` Statement

Let's expand our example a little. What if we want to display a different message if it is 11 a.m. or later? Can we do it using only the tools we have? Of course we can!

```
if (hourOfDay < 11) {  
    System.out.println("Good Morning");  
}
```

```
if (hourOfDay >= 11) {  
    System.out.println("Good Afternoon");  
}
```

This seems a bit redundant, though, since we're performing an evaluation on `hourOfDay` twice. Luckily, Java offers us a more useful approach in the form of an `else` statement, as shown in [Figure 3.2](#).

 A sample structure of an `else` statement. It reads `if` as `if` keyword, boolean expression as parenthesis, braces as a block of multiple statements of optional for single statement, and `else` as optional `else` keyword.

[FIGURE 3.2](#) The structure of an `else` statement

Let's return to this example:

```
if (hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else System.out.println("Good Afternoon");
```

Now our code is truly branching between one of the two possible options, with the boolean evaluation happening only once. The `else` operator takes a statement or block of statements, in the same manner as the `if` statement. Similarly, we can append additional `if` statements to an `else` block to arrive at a more refined example:

```
if (hourOfDay < 11) {  
    System.out.println("Good Morning");  
} else if (hourOfDay < 15) {  
    System.out.println("Good Afternoon");  
} else {  
    System.out.println("Good Evening");  
}
```

In this example, the Java process will continue execution until it encounters an `if` statement that evaluates to `true`. If neither of the first two expressions is `true`, it will execute the code of the final `else` block.

Verifying That the *if* Statement Evaluates to a Boolean Expression

Another common way the exam may try to lead you astray is by providing code where the boolean expression inside the `if` statement is not actually a boolean expression. For example, take a look at the following lines of code:

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

This statement may be valid in some other programming and scripting languages, but not in Java, where `0` and `1` are not considered boolean values.

Shortening Code with Pattern Matching

Pattern matching is a technique of controlling program flow that only executes a section of code that meets certain criteria. In this section, we perform pattern matching with the `if` statement, along with the `instanceof` operator, to improve program control.



If pattern matching is new to you, be careful not to confuse it with the Java `Pattern` class or regular expressions (*regex*). While pattern matching can include the use of regular expressions for filtering, they are unrelated concepts.

Pattern matching is a useful tool for reducing boilerplate code in your application. *Boilerplate code* is code that tends to be duplicated throughout

a section of code over and over again in a similar manner.

To understand why this feature was added, consider the following code that takes a `Number` instance and compares it with the value 5. If you haven't seen `Number` or `Integer`, you just need to know that `Integer` inherits from `Number` for now. You'll see them a lot in this book!

```
void compareIntegers(Number number) {
    if (number instanceof Integer) {
        Integer data = (Integer)number;
        System.out.print(data.compareTo(5));
    }
}
```

The cast is needed since the `compareTo()` method is defined on `Integer`, but not on `Number`.

Code that first checks if a variable is of a particular type and then immediately casts it to that type is extremely common in the Java world. It's so common that the authors of Java decided to implement a shorter syntax for it:

```
void compareIntegers(Number number) {
    if (number instanceof Integer data) {
        System.out.print(data.compareTo(5));
    }
}
```

The variable `data` in this example is referred to as the *pattern variable*. Notice that this code also avoids any potential `ClassCastException` because the cast operation is executed only if the `instanceof` operator returns true.

[Figure 3.3](#) shows the anatomy of pattern matching using the `instanceof` operator and `if` statements. Adding a variable after the type is what instructs the compiler to treat it as pattern matching. [Figure 3.3](#) also shows an optional conditional clause, which is a useful feature that we will cover in the next section.



A sample structure depicts the anatomy of pattern matching. It reads tickets as input, instanceof as pattern matching operator, integer as type, I as pattern variable, and two and symbol d lesser than 10 as optional conditional clause.

FIGURE 3.3 Pattern matching with if

Reassigning Pattern Variables

While possible, it is a bad practice to reassign a pattern variable since doing so can lead to ambiguity about what is and is not in scope.

```
if (number instanceof Integer data) {  
    data = 10;  
}
```

The reassignment can be prevented with a `final` modifier, but it is better not to reassign the variable at all.

```
if (number instanceof final Integer data) {  
    data = 10; // DOES NOT COMPILE  
}
```

Pattern Variables and Expressions

Pattern matching supports an optional conditional clause, declared as a boolean expression. This can be used to filter data out, such as in the following example:

```
void printIntegersGreaterThan5(Number number) {  
    if (number instanceof Integer data && data.compareTo(5) > 0)  
        System.out.print(data);  
}
```

We can apply a number of filters, or patterns, so that the `if` statement is executed only in specific circumstances. Notice that we're using the pattern variable in an expression in the same line in which it is declared.

Pattern Matching with *null*

As saw in [Chapter 2](#), “Operators,” the instanceof operator always evaluates null references to false. The same holds for pattern matching.

```
String noObjectHere = null;

if(noObjectHere instanceof String)
    System.out.println("Not printed");

if(noObjectHere instanceof String s)
    System.out.println("Still not printed");

if(noObjectHere instanceof String s && s.length() > -1)
    System.out.println("Nope, not this one either");
```

As shown in the last example, this also helps avoid any potential NullPointerException, as the conditional operator (&&) causes the s.length() call to be skipped.

Supported Types

The type of the pattern variable must be a compatible type, which includes the same type, a subtype, or a supertype of the reference variable. If the reference variable does not refer to a final class or type, then it can also include an unrelated interface for reasons we’ll go into more detail about in [Chapter 7](#), “Beyond Classes.” Consider the following two examples, in which Integer is a subtype of Number:

```
11: Number bearHeight = Integer.valueOf(123);
12:
13: if (bearHeight instanceof Integer i) {}
14: if (bearHeight instanceof Number n) {}
15: if (bearHeight instanceof String s) {} // DOES NOT COMPILE
16: if (bearHeight instanceof Object o) {}
```

The first example uses a subtype, while the second example uses the same type as the reference variable bearHeight. On line 15, the compiler recognizes that a Number cannot be cast to an unrelated type String and throws an error. Line 16 is permitted but not particularly useful, since every Object except null will return true.



When pattern matching was first introduced in Java, the type had to be a strict subtype of `Number` (and not `Number` itself). For this reason, lines 14 and 16 would not compile. Starting with Java 21, this rule was removed to allow the same or broader types to be used.

Flow Scoping

The compiler applies flow scoping when working with pattern matching. *Flow scoping* means the variable is only in scope when the compiler can definitively determine its type. *Flow scoping is unlike any other type of scoping*, in that it is not strictly hierarchical. It is determined by the compiler based on the branching and flow of the program.

Given this information, can you see why the following does not compile?

```
void printIntegersOrNumbersGreaterThan5(Number number) {  
    if (number instanceof Integer data || data.compareTo(5) > 0)  
        System.out.print(data);  
}
```

The key thing to notice is that we used OR (`||`) not AND (`&&`) in the conditional statement. If the input does not inherit `Integer`, the `data` variable is undefined. Since the compiler cannot guarantee that `data` is an instance of `Integer`, the code does not compile.

What about this example?

```
void printIntegerTwice(Number number) {  
    if (number instanceof Integer data)  
        System.out.print(data.intValue());  
    System.out.print(data.intValue()); // DOES NOT COMPILE  
}
```

Since the input might not have inherited `Integer`, `data` is no longer in scope after the `if` statement. Oh, so you might be thinking that the pattern variable is then only in scope inside the `if` statement block, right? Well, not exactly! Consider the following example that does compile:

```

void printOnlyIntegers(Number number) {
    if (!(number instanceof Integer data))
        return;
    System.out.print(data.intValue());
}

```

It might surprise you to learn this code does compile. Eek! What is going on here? The method returns if the input does not inherit Integer. This means that when the last line of the method is reached, the input must inherit Integer, and therefore data stays in scope even after the if statement ends. Understanding why this example compiles and the one before it does not, is the key to understanding flow scoping.

Flow Scoping and *else* Branches

If the last code sample confuses you, don't worry: you're not alone! Another way to think about it is to rewrite the logic to something equivalent that uses an else statement:

```

void printOnlyIntegers(Number number) {
    if (!(number instanceof Integer data))
        return;
    else
        System.out.print(data.intValue());
}

```

We can now go one step further and reverse the if and else branches by inverting the boolean expression:

```

void printOnlyIntegers(Number number) {
    if (number instanceof Integer data)
        System.out.print(data.intValue());
    else
        return;
}

```

Our new code is equivalent to our original and better demonstrates how the compiler was able to determine that data was in scope only when number is an Integer.

Make sure you understand the way flow scoping works. In particular, it is possible to use a pattern variable outside of the `if` statement, but only when the compiler can definitively determine its type.



But wait, there's more pattern matching fun coming! In the next section, we'll use pattern matching with `switch` statements, and in [Chapter 7](#), we'll apply pattern matching to records and sealed classes.

Building *switch* Statements and Expressions

An `if/else` statement can get really difficult to read if there are a lot of branches. Take a look at the following:

```
String getAnimalBad(int type) {  
    String animal;  
    if (type == 0)  
        animal = "Lion";  
    else if (type == 1)  
        animal = "Elephant";  
    else if (type == 2 || type == 3)  
        animal = "Alligator";  
    else if (type == 4)  
        animal = "Crane";  
    else  
        animal = "Unknown";  
    return animal;  
}
```

Every time we add a new animal, that code gets longer and more difficult to maintain. Luckily, Java includes `switch` to help simplify this code.

Introducing *switch*

A `switch` is a complex decision-making structure in which a single value is evaluated and flow is redirected to one or more branches. In Java, there are two flavors: a `switch` statement and a `switch` expression. The primary

difference between the two (aside from a lot of syntax differences!) is that a switch expression must return a value, while a switch statement does not.

Let's begin by rewriting our previous method to one that uses a switch statement.

```
String getAnimalBetter(int type) {  
    String animal;  
    switch (type) {  
        case 0:  
            animal = "Lion";  
            break;  
        case 1:  
            animal = "Elephant";  
            break;  
        case 2, 3:  
            animal = "Alligator";  
            break;  
        case 4:  
            animal = "Crane";  
            break;  
        default:  
            animal = "Unknown";  
    }  
    return animal;  
}
```

That's certainly better than our if/else version in terms of keeping things organized, but it's still really long. We're assigning animal four times, and what's with all the break statements? We'll cover all of these details shortly, but for now let's try using a switch expression instead.

```
String getAnimalBest(int type) {  
    return switch (type) {  
        case 0    -> "Lion";  
        case 1    -> "Elephant";  
        case 2, 3 -> "Alligator";  
        case 4    -> "Crane";  
        default   -> "Unknown";  
    };  
}
```

Wow, that is a lot shorter and easier to read! In Java, switch expressions were introduced more recently than switch statements, resulting in a greater emphasis on reducing boilerplate.



As you might remember from [Chapter 1](#), extra whitespace doesn't matter in Java. That said, whitespace can be used to align text and help improve readability. When writing many of the examples in this chapter, we added extra whitespace to make switch statements and expressions easier to read.

Structuring *switch* Statements and Expressions

While switch statements and expressions may look different, they share many common rules that we cover in this section. Afterward, we'll cover the specifics of each type.

Defining a *switch*

First off, both types start with a switch keyword and a variable wrapped in parentheses.

```
String name = "123";
```

```
switch (name) {                                // Switch statement
    case "Sancha":      System.out.print(1);    break;
    case "Jacob", "Jake": System.out.print(2);  break;
    default:           System.out.print(999);  break;
}
```

```
System.out.println(switch (name) { // Switch expression
    case "Sancha"      -> 1;
    case "Jacob", "Jake" -> 2;
    default            -> 999;
});
```

As you can see, both types of switch support zero or more case clauses. Each case clause includes a set of matching values split up by commas (,). It is then followed by a separator, which can be a colon (:) or the arrow operator (->). Finally, each clause then defines an expression, or code block with braces ({}), for what to execute when there's a match.

Using the Arrow Operator with *switch* Statements

While *switch* statements support both colons and arrow operators, you're likely to see them used with colons more often in practice. This is because the colon syntax has been around a lot longer in Java. If you do use the arrow operator, then you must use it for all clauses. For example, the following *switch* statement does not compile:

```
switch (type) {  
    case 0 : System.out.print("Lion");  
    case 1 -> System.out.print("Elephant");  
}
```

This would compile if both clauses used the same operator.

Both *switch* types support an optional default clause. With *switch* expressions, a default clause is often required, as the expression must return a value. More on that shortly.

Without further ado, [Figure 3.4](#) shows the structure of a *switch* statement. While *switch* statements can use the arrow operator, they are more commonly written with colons.



A sample structure of a *switch* statement. It reads *switch* as *switch* keyword, variable to test as parenthesis, *break* as optional *break*, open braces as beginning curly brace, close brace as ending curly brace, and *default* as optional default that may appear anywhere within *switch* statement.

FIGURE 3.4 A *switch* statement

[Figure 3.5](#) shows the structure of a *switch* expression, which returns a value that is assigned to a variable. Compare it with [Figure 3.4](#) and make sure you understand the differences between the two.

Unlike a *switch* statement, a *switch* expression often requires a semicolon (;) after it, such as when it is used with the assignment operator (=) or a

return statement. This has more to do with *how* the switch expression is used than the switch expression itself.

A switch expression also requires a semicolon (;) after each case expression that doesn't use a block. For example, how many semicolons are missing in the following?

```
var result = switch (bear) {  
    case 30 -> "Grizzly"  
    default -> "Panda"  
}
```

The answer is three. Each case or default expression requires a semicolon as well as the assignment itself. The following fixes the code:

```
var result = switch (bear) {  
    case 30 -> "Grizzly";  
    default -> "Panda";  
};
```



A sample structure of a switch expression. It reads int result as assign expression to value, switch as switch keyword, variable to test as parenthesis, break as optional break, open braces as beginning curly brace, close brace as ending curly brace, semicolon required if used with the assignment operator, and a default branch may appear anywhere within the switch expression and is required if all possible values are not handled.

FIGURE 3.5 A switch expression

Challenge time! See if you can figure out which of these compiles:

```
int food = 5, month = 4, weather = 2, day = 0, time = 4;
```

```
String meal = switch food { // #1  
    case 1 -> "Dessert"  
    default -> "Porridge"  
};
```

```
switch (month) // #2  
    case 4: System.out.print("January");
```

```
switch (weather) { // #3  
    case 2: System.out.print("Rainy");  
    case 5: {  
        System.out.print("Sunny");
```



```

    }
}

switch (day) { // #4
    case 1: 13: System.out.print("January");
    default System.out.print("July");
}

String description = switch (time) { // #5
    case 10 -> "Morning";
    default -> "Late";
}

```

The first statement does not compile because the switch expression is missing parentheses around the switch variable, as well as semicolons after the case and default clauses. The second statement does not compile because it is missing braces around the switch body. The third statement does compile. Notice that a case clause can use an expression or a code block with braces.

The fourth statement does not compile for two reasons. The case clause should use a comma (,) to separate two values, not a colon (:). It's also missing a colon after the default clause. The last statement does not compile because the assignment operator (=) is missing a semicolon (;) at the end.



A switch statement is not required to contain any case clauses. This is perfectly valid:

```
switch (month) {}
```

Selecting the *switch* Variable

As shown in [Figures 3.4](#) and [3.5](#), a switch has a target variable that is not evaluated until runtime. The following is a list of all data types supported by switch:

- `int` and `Integer`
- `byte` and `Byte`
- `short` and `Short`
- `char` and `Character`
- `String`
- `enum` values
- All object types (when used with pattern matching)
- `var` (if the type resolves to one of the preceding types)



Notice that `boolean`, `long`, `float`, and `double` are not supported in `switch` statements and expressions.

If you've never worked with enums, don't panic! For this chapter, you just need to know that an enumeration, or *enum*, is a type in Java that represents a fixed set of constants, such as the following:

```
enum Season { SPRING, SUMMER, FALL, WINTER }
```

```
enum DayOfWeek {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,  
    SATURDAY  
}
```

If it helps, think of an enum as a class type with predefined object values known at compile time. We cover enums in detail in [Chapter 7](#), including showing how they can define variables, methods, and constructors. For now, you just need to think of them as a list of values.

Starting with Java 21, `switch` statements and expressions now support pattern matching, which means *any object type* can now be used as a

switch variable, provided the pattern matching is used. We'll be covering switch with pattern matching shortly.

Determining Acceptable Case Values

Not just any values can be used in a case clause. First, the values in each case clause must be *compile-time constant values*. This means you can use only literals, enum constants, or final constant variables.

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. For example, you can't have a case clause value that requires executing a method at runtime, even if that method always returns the same value.

For these reasons, see if you can figure out why only the first and last case clauses compile:

```
final int getCookies() { return 4; }
void feedAnimals() {
    final int bananas = 1;
    int apples = 2;
    int numberOfAnimals = 3;
    final int cookies = get_cookies();
    switch (numberOfAnimals) {
        case bananas:
        case apples:           // DOES NOT COMPILE
        case get_cookies():   // DOES NOT COMPILE
        case cookies :        // DOES NOT COMPILE
        case 3 * 5 :
    } }
```

The bananas variable is marked final, and its value is known at compile time, so the first case clause is valid. The apples variable in the second case clause is not marked final, so it is not permitted. The next two case clauses, with values get_cookies() and cookies, do not compile because methods are not evaluated until runtime, so they cannot be used as the value of a case clause, even if one of the values is stored in a final variable. The last case clause, with value 3 * 5, does compile, as expressions are allowed as case values, provided the value can be resolved at compile time.

Remember, the data type for case clauses must match the data type of the switch variable. See if you can spot why the following does not compile:

```
String cleanFishTank(int dirty) {
    return switch (dirty) {
        case "Very" -> "1 hour"; // DOES NOT COMPILE
        default     -> "45 minute";
    };
}
```

The switch variable is of type `int`, while the case clause is of type `String`.

Using Enum Values with *switch*

When the switch variable is an enum type, then the case clauses must be the enum values.

```
enum Season { SPRING, SUMMER, FALL, WINTER }

boolean shouldGetACoat(Season s) {
    return switch (s) {
        case SPRING -> false;
        case Season.SUMMER -> false;
        case FALL -> true;
        case Season.WINTER -> true;
    };
}
```

For an enum value, you can specify just the value, as shown with `SPRING` and `FALL`. Starting with Java 21, you can optionally specify the name with the value, such as `Season.SPRING` and `Season.FALL`.

Working with *switch* Statements

Taking a look at the earlier `getAnimalBetter()` method, you might remember there were a lot of `break` statements at the end of each case. A `break` statement terminates the switch statement and returns flow control to the enclosing process. Put simply, it ends the switch statement immediately.

While `break` statements are optional, they tend to be used frequently in switch statements. Without `break` statements, the code continues to execute the next branch it finds, in order.

What do you think the following prints when `printSeasonForMonth(2)` is called?

```
void printSeasonForMonth(int month) {  
    switch (month) {  
        case 1, 2, 3:    System.out.print("Winter-");  
        case 4, 5, 6:    System.out.print("Spring-");  
        default:         System.out.print("Unknown-");  
        case 7, 8, 9:    System.out.print("Summer-");  
        case 10, 11, 12: System.out.print("Fall-");  
    } }  
}
```

It prints everything!

Winter-Spring-Unknown-Summer-Fall-

It matches the first case clause and executes all of the branches in the order they are found, including the default clause. Remember when working with switch statements, the *order of the branches is important!* We can fix this to just print winter- with the same input, by adding break statements.

```
void printSeasonForMonth(int month) {  
    switch (month) {  
        case 1, 2, 3:    System.out.print("Winter-");    break;  
        case 4, 5, 6:    System.out.print("Spring-");    break;  
        default:         System.out.print("Unknown-");    break;  
        case 7, 8, 9:    System.out.print("Summer-");    break;  
        case 10, 11, 12: System.out.print("Fall-");      break;  
    } }  
}
```

The last case clause does not actually require a break, as the switch statement is over, but we add it for consistency.

The exam creators are fond of switch examples that are missing break statements! When you spot a switch statement on the exam, always consider whether multiple branches may be visited in a single execution.

Contrast this with a switch expression that matches only a single branch at runtime and therefore does not require break statements.

```
void printSeasonForMonth(int month) {  
    String value = switch (month) {  
        case 1, 2, 3    -> "Winter-";  
        case 4, 5, 6    -> "Spring-";  
        default         -> "Unknown-";  
        case 7, 8, 9     -> "Summer-";  
        case 10, 11, 12 -> "Fall-";  
    };  
    System.out.print(value);  
}
```

When Is a *switch* Expression Not a *switch* Expression?

As stated earlier, a switch expression always returns a value, regardless of the syntax used. What about the following?

```
void printWeather(int rain) {  
    switch (rain) {  
        case 0 -> System.out.print("Dry");  
        case 1 -> System.out.print("Wet");  
        case 2 -> System.out.print("Storm");  
    }  
}
```

Since the return type of `System.out.print()` is `void`, this statement does not return a value. This is actually a switch statement that uses the arrow operator syntax. Since it doesn't return a value, it is not a switch expression. It's a little confusing, we know!

Working with *switch* Expressions

Congratulations, you're now an expert on switch statements!

Unfortunately, switch expressions have a lot more features and rules we still need to cover. We cover them (one at a time) in this section.

Returning Consistent Data Types

Just as case values have to use a consistent data type, the switch expression must return a consistent value. Simply put, when assigning a value as the result of a switch expression, a branch can't return a value with an unrelated type.

```
int measurement = 10;
int size = switch (measurement) {
    case 5  -> Integer.valueOf(1);
    case 10 -> (short)2;
    default -> 3;
    case 20 -> "4";    // DOES NOT COMPILE
    case 40 -> 5L;     // DOES NOT COMPILE
    case 50 -> null;   // DOES NOT COMPILE
};
```

The switch expression is being assigned to an int variable, so all of the values must be consistent with int. The first case clause compiles without issue, as the Integer value is unboxed to int. We'll cover autoboxing and unboxing in [Chapter 5](#), "Methods." The second and third case clauses are fine, as they can be stored as an int. The last three case expressions do not compile because each returns a type that is incompatible with int.

Exhausting the switch Branches

Unlike a switch statement, a switch expression must return a value. Why? Let's try an illustrative example.

```
void identifyType(String type) {
    Integer reptile = switch (type) { // DOES NOT COMPILE
        case "Snake"  -> 1;
        case "Turtle" -> 2;
    };
}
```

What is the value of reptile if type is not equal to Snake or Turtle? Does it throw an exception? Return null or -1? The answer is "None of the above." Java decided this behavior would be unsupported and triggers a compiler error if the switch expression is not exhaustive.

A switch is said to be *exhaustive* if it covers all possible values. All switch expressions must be exhaustive, which means they must handle all possible

values. As we'll see shortly, there are times a switch statement must be exhaustive too. There are three ways to write an exhaustive switch:

1. Add a default clause.
2. If the switch takes an enum, add a case clause for every possible enum value.
3. Cover all possible types of the switch variable with pattern matching.

In practice, the first solution is the one most often used. You can try writing case clauses for all possible int values, but we promise it doesn't work! Even smaller types like byte are not permitted by the compiler, despite there being only 256 possible values.

The second solution applies only to switch expressions that take an enum. For example, consider the following:

```
enum Season { SPRING, SUMMER, FALL, WINTER }

String getWeatherMissingOne(Season value) {
    return switch (value) { // DOES NOT COMPILE
        case WINTER -> "Cold";
        case SPRING -> "Rainy";
        case SUMMER -> "Hot";
    };
}
```

This code does not compile because FALL is not covered. The fix is either to add a case for FALL or add a default clause (or both).

```
String getWeatherCoveredAll(Season value) {
    return switch (value) {
        case WINTER -> "Cold";
        case SPRING -> "Rainy";
        case SUMMER -> "Hot";
        case FALL -> "Warm";
        default -> throw new RuntimeException("Unsupported
Season");
    };
}
```

Since all possible values of Season are covered, the default branch is optional.

When writing switch expressions, it may be a good idea to add a default branch, even if you cover all possible values. This means that if someone modifies the enum with a new value, your code will still compile.

The third solution for writing an exhaustive switch requires some explanation. We'll cover this in an upcoming section, as it only applies to pattern matching.

Using the *yield* Statement

Up until now, the switch expression examples we've shown use case expressions. Time to expand your knowledge to case blocks! A switch expression supports both case expressions and case blocks, the latter of which is denoted with braces (`{}`). [Figure 3.6](#) shows examples of both.



A sample structure of a switch expression with a case block and yield statement. It reads case expression, case block, curly braces required for case blocks, and a yield is required within a case block unless an exception is thrown.

FIGURE 3.6 A switch expression with a case block and yield statement

In the previous section, we said that a switch expression must return a value. But how do you return a value from a case block? You could use a return statement, but that ends the method, not just the switch expression!

Enter the yield statement, shown in [Figure 3.6](#). It allows the case clause to return a value. For example, the following uses a mix of case expressions and case blocks:

```
int fish = 5;
int length = 12;
var name = switch (fish) {
    case 1 -> "Goldfish";
    case 2 -> { yield "Trout"; }
    case 3 -> {
        if (length > 10) yield "Blobfish";
        else yield "Green";
    }
}
```

```

    }
    case 4 -> {
        throw new RuntimeException("Unsupported value");
    }
    default -> "Swordfish";
};

```

Think of the `yield` keyword as a return statement within a switch expression. Because a switch expression must return a value, a `yield` is often required within a case block. The one “exception” to this rule is if the code throws an exception, as shown in the previous example.

Watch Semicolons in *switch* Expressions

When writing a case expression, a semicolon is required, but when writing a case block, it is prohibited.

```

int fish = 1;
var name = switch (fish) {
    case 1 -> "Goldfish" // DOES NOT COMPILE
(missing semicolon)
    case 2 -> { yield "Trout"; }; // DOES NOT COMPILE
(extra semicolon)
    default -> "Shark";
} // DOES NOT COMPILE (missing semicolon)

```

A bit confusing, right? It’s just one of those things you have to train yourself to spot on the exam.

Using Pattern Matching with *switch*

One of the biggest new features of Java 21 is that pattern matching has been extended to `switch`. There’s a number of rules to cover, so let’s start with the basics. To use pattern matching with a `switch`, first start with an object reference variable. *Any object reference type is permitted*, provided the `switch` makes use of pattern matching. Next, in each case clause, define a type and pattern matching variable.

```

void printDetails(Number height) {
    String message = switch (height) {
        case Integer i -> "Rounded: " + i;
        case Double d   -> "Precise: " + d;
        case Number n   -> "Unknown: " + n;
    };
    System.out.print(message);
}

```

In this example, we output different values depending on the type of the switch variable. The same rules about local variables and flow scoping that we learned about earlier with pattern matching apply. For instance, the pattern matching variable exists only within the case branch for which it is defined. This allows us to reuse the same name for two case branches.

[Figure 3.7](#) shows the structure of pattern matching with a switch expression. It can also be used with a switch statement that does not return a value.

Easy so far, right? From [Figure 3.7](#), you might have noticed it includes a *guard clause*, which is an optional conditional clause that can be added to a case branch. This is similar to what we saw in [Figure 3.3](#) with a pattern matching if statement. The only difference is that with switch, the `when` keyword is required between the variable and the expression.

Let's try an example. Suppose our zoo has different trainers that can handle different size animals depending on the measurement type.

```

String getTrainer(Number height) {
    return switch (height) {
        case Integer i when i > 10 -> "Joseph";
        case Integer i -> "Daniel";
        case Double num when num <= 15.5 -> "Peter";
        case Double num -> "Kelly";
        case Number num -> "Ralph";
    };
}

```

In this example, Joseph works with the animal if the height is an Integer greater than 10. Daniel is then selected for all other Integer values. Likewise, Peter handles all Double measurements less than or equal to 15.5, while Kelly handles the remaining Double values. Finally, Ralph

handles all animals that don't meet one of the previous requirements, such as if `Short` was used.


 A sample structure of pattern matching with a switch expression. It reads integer as type, `x` as pattern variable, when `x` greater than zero and `x` lesser than or equal to twenty as guard, and case number as type matches switch variable reference type so default clause is not needed.

FIGURE 3.7 Pattern matching with switch

One advantage of guards is that now switch can do something it's never done before: *it can handle ranges*. Previously, if you wanted to support a range of values with a switch, you had to list all the possible case values. With the `when` clause, you can support range matches. Quite convenient!

Applying Acceptable Types

One of the simplest rules when working with switch and pattern matching is that the type can't be unrelated. It must be the same type as the switch variable or a subtype.

```
Number fish = 10;
String name = switch (fish) {
    case Integer freshWater -> "Bass";
    case Number saltWater   -> "ClownFish";
    case String s           -> "Shark"; // DOES NOT COMPILE
};
```

The compiler is smart enough to know a `Number` can't be cast as a `String`, resulting in this code not compiling. This wasn't allowed in the previous pattern matching section using `instanceof` either!

Ordering switch Branches

As we mentioned earlier in the chapter, the order of case and default clauses for switch statement matters, because more than one branch might be reached during execution. For switch expressions that don't use pattern matching, ordering isn't important, as only one branch can be reached.

Well, when working with pattern matching, the order matters regardless of the type of switch! For example, consider this new version of `printDetails()` in which the order has been changed:

```

void printDetails(Number height) {
    String message = switch (height) {
        case Number n    -> "Unknown: " + n;
        case Integer i    -> "Rounded: " + i;
        case Double d     -> "Precise: " + d;
    };
    System.out.print(message);
}

```

The code no longer compiles as the second and third case clauses are considered *dominated* by the preceding case `Number` statement. To put it another way, it is impossible for any process to reach these two case clauses. This is also referred to as *unreachable code*, which we cover more later in this chapter. In most cases, when the compiler detects unreachable code, it results in a compiler error.

Ordering branches is also important if a `when` clause is used. For example, what if we reordered the first two branches of our `getTrainer()` method?

```

String getTrainer(Number animal) {
    return switch (animal) {
        case Integer i                -> "Daniel";
        case Integer i when i > 10    -> "Joseph"; // DOES NOT
COMPILE
        ...
    };
}

```

In the event that `animal` is an `Integer`, `Daniel` will always be selected. Poor Joseph! Likewise, the compiler does not allow this code to compile.

Exhaustive *switch* Statements

Up until now, only `switch` expressions were required to be exhaustive. When using pattern matching, `switch` statements must be exhaustive too. As before, you can address this with a `default` clause, as this will handle all values that don't match a case clause. There is another option, though, that applies only when working with pattern matching.

You may have noticed that we didn't use any `default` clauses in any of our previous pattern matching examples. That's because we defined our last case clause with a pattern matching variable type that is *the same as the switch variable reference type*.

That might sound complicated, but it's simpler than it seems. For example, if the variable reference type of the switch expression is type Integer or String, then you just need to make sure the last case clause is of type Integer or String, respectively.

Let's try an illustrative example. What do you expect the output of the following to be?

```
Number zooPatrons = Integer.valueOf(1_000);
switch (zooPatrons) {
    case Integer count -> System.out.print("Welcome: " + count);
}
```

It doesn't compile! Despite the zooPatrons object actually being of type Integer, the switch reference variable is of type Number. There are a few ways that we can fix this. First, we can change the reference type of zooPatrons to be Integer, which results in all possible values of Integer being covered.

```
Integer zooPatrons = Integer.valueOf(1_000);
switch (zooPatrons) {
    case Integer count -> System.out.print("Welcome: " + count);
}
```

Alternatively, we can also add a case clause at the end for Number.

```
Number zooPatrons = Integer.valueOf(1_000);
switch (zooPatrons) {
    case Integer count -> System.out.print("Welcome: " + count);
    case Number count -> System.out.print("Too many people at
the zoo!");
}
```

There is a third option, too! Don't forget, we can always add a default clause to a switch that covers everything.

That brings us to an interesting question: what if you combine different solutions?

```
Number zooPatrons = Integer.valueOf(1_000);
switch (zooPatrons) {
    case Integer count -> System.out.print("Welcome: " + count);
    case Number count -> System.out.print("Too many people at
the zoo!");
}
```

```
    default                -> System.out.print("The zoo is closed");  
}
```

In this case, the code does not compile, regardless of how you order the branches. The compiler is smart enough to realize the last two statements are redundant, as one always dominates the other.

Handling a *null* Case

What if the switch variable is null at runtime? We can try using a default clause but you might be surprised at the result.

```
String fish = null;  
System.out.print(switch (fish) {  
    case "ClownFish" -> "Hello!";  
    case "BlueTang"  -> "Hello again!";  
    default          -> "Goodbye";  
});
```

This code compiles (it technically is exhaustive) but throws a `NullPointerException`! One “quick fix” would be to add an if/else statement around the switch, but that would add a lot of extra boilerplate code.

```
String fish = null;  
if (fish == null) {  
    System.out.print("What type of fish are you?");  
} else {  
    System.out.print(switch (fish) {  
        case "ClownFish" -> "Hello!";  
        case "BlueTang"  -> "Hello again!";  
        default          -> "Goodbye";  
    });  
}
```

New to Java 21, switch now supports case null clause when working with object types, allowing us to rewrite our previous example as the following:

```
String fish = null;  
System.out.print(switch (fish) {  
    case "ClownFish" -> "Hello!";  
    case "BlueTang"  -> "Hello again!";  
    case null        -> "What type of fish are you?";  
    default          -> "Goodbye";  
});
```

That's a lot less boilerplate code, now that we don't have to handle `null` separately.

Case *null* Is Considered Pattern Matching

Any guess as to why the following code snippet does not compile?

```
String fish = null;
switch (fish) { // DOES NOT COMPILE
    case "ClownFish": System.out.print("Hello!");
    case "BlueTang":  System.out.print("Hello again!");
    case null:        System.out.print("What type of fish are
you?");
}
```

Anytime `case null` is used within a switch, then the switch statement is considered to use pattern matching. As you should remember from the previous section, that means the switch statement *must be exhaustive*. Adding a default branch allows the code to compile.

Since using `case null` implies pattern matching, the ordering of branches matters anytime `case null` is used. While `case null` can appear almost anywhere in switch, it cannot be used after a default statement. For instance, only the first of the following two switch statements compile.

```
System.out.print(switch (fish) {
    case String s when "ClownFish".equals(s) -> "Hello!";
    case null -> "No good";
    case String s when "BlueTang".equals(s) -> "Hello again!";
    default -> "Goodbye";
});
System.out.print(switch (fish) {
    case String s when "ClownFish".equals(s) -> "Hello!";
    case String s when "BlueTang".equals(s) -> "Hello again!";
    default -> "Goodbye";
    case null -> "No good"; // DOES NOT COMPILE
});
```

In the second example, the default clause dominates the `case null` clause. For the exam, make sure you can identify where default and `case null`

can be used within a switch.

Writing *while* Loops

A common practice when writing software is doing the same task some number of times. You could use the decision structures we have presented so far to accomplish this, but that's going to be a pretty long chain of `if` or `else` statements, especially if you have to execute the same thing 100 times or more.

Enter loops! A *loop* is a repetitive control structure that can execute a statement or block of code multiple times in succession. By using variables that can be assigned new values, each repetition of the statement may be different. The following loop executes exactly 10 times:

```
int counter = 0;
while (counter < 10) {
    double price = counter * 10;
    System.out.println(price);
    counter++;
}
```

If you don't follow this code, don't panic—we cover it shortly. In this section, we're going to discuss the `while` loop and its two forms. In the next section, we move on to `for` loops, which have their roots in `while` loops.

The *while* Statement

The simplest repetitive control structure in Java is the `while` statement, described in [Figure 3.8](#). Like all repetition control structures, it has a *termination condition*, implemented as a boolean expression. A loop will continue as long as this expression evaluates to `true`.

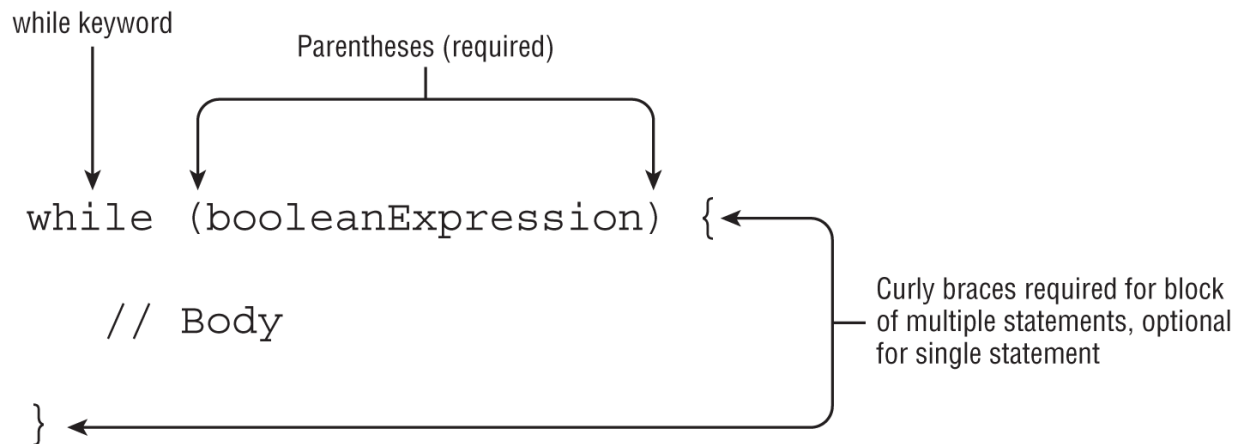


FIGURE 3.8 The structure of a while statement

As shown in [Figure 3.8](#), a while loop is similar to an if statement in that it is composed of a boolean expression and a statement, or a block of statements. During execution, the boolean expression is evaluated before each iteration of the loop and exits if the evaluation returns false.

Let's see how a loop can be used to model a mouse eating a meal:

```
int roomInBelly = 5;
void eatCheese(int bitesOfCheese) {
    while (bitesOfCheese > 0 && roomInBelly > 0) {
        bitesOfCheese--;
        roomInBelly--;
    }
    System.out.println(bitesOfCheese+" pieces of cheese left");
}
```

This method takes an amount of food—in this case, cheese—and continues until the mouse has no room in its belly or there is no food left to eat. With each iteration of the loop, the mouse “eats” one bite of food and loses one spot in its belly. By using a compound boolean statement, you ensure that the while loop can end for either of the conditions.

One thing to remember is that a while loop may terminate after its first evaluation of the boolean expression. For example, how many times is Not full! printed in the following example?

```
int full = 5;
while (full < 5) {
    System.out.println("Not full!");
}
```

```

    full++;
}

```

The answer? Zero! On the first iteration of the loop, the condition is reached, and the loop exits. This is why *while* loops are often used in places where you expect zero or more executions of the loop. Simply put, the body of the loop may not execute at all.

The *do/while* Statement

The second form a *while* loop can take is called a *do/while* loop, which, like a *while* loop, is a repetition control structure with a termination condition and statement, or a block of statements, as shown in [Figure 3.9](#).

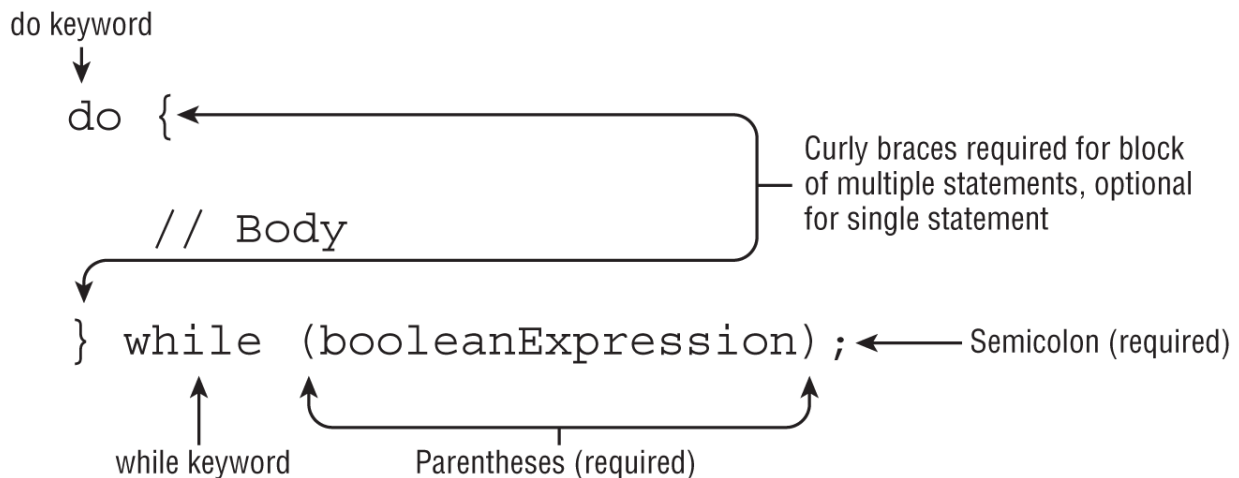


FIGURE 3.9 The structure of a *do/while* statement

Unlike a *while* loop, though, a *do/while* loop guarantees that the statement or block will be executed *at least once*. For example, what is the output of the following statements?

```

int lizard = 0;
do {
    lizard++;
} while (false);
System.out.println(lizard);

```

Java will execute the statement block first and then check the loop condition. Even though the loop exits right away, the statement block is still executed once, and the program prints 1.

Infinite Loops

The single most important thing you should be aware of when you are using any repetition control structures is to make sure they always terminate! Failure to terminate a loop can lead to numerous problems in practice, including overflow exceptions, memory leaks, slow performance, and even bad data. Let's take a look at an example:

```
int pen = 2;  
int pigs = 5;  
while (pen < 10)  
    pigs++;
```

You may notice one glaring problem with this statement: it will never end. The variable `pen` is never modified, so the expression `(pen < 10)` will always evaluate to `true`. The result is that the loop will never end, creating what is commonly referred to as an infinite loop. An *infinite loop* is a loop whose termination condition is never reached during runtime.

Anytime you write a loop, you should examine it to determine whether the termination condition is always eventually met under some condition. For example, a loop in which no variables are changing between two executions suggests that the termination condition may not be met. The loop variables should always be moving in a particular direction.

In other words, make sure the loop condition, or the variables the condition is dependent on, are changing between executions. Then, ensure that the termination condition will be eventually reached in all circumstances. As you learn in the last section of this chapter, a loop may also exit under other conditions, such as a `break` or `return` statement.

Constructing *for* Loops

Even though `while` and `do/while` statements are quite powerful, some tasks are so common in writing software that special types of loops were created—for example, iterating over a statement exactly 10 times or iterating over a list of names. You could easily accomplish these tasks with various `while` loops that you've seen so far, but they usually require a lot of boilerplate code. Wouldn't it be great if there was a looping structure that could do the same thing in a single line of code?

With that, we present the most convenient repetition control structure, for loops. There are two types of for loops, although both use the same for keyword. The first is referred to as the *basic* for loop, and the second is often called the *enhanced* for loop. For clarity, we refer to them as the for loop and the for-each loop, respectively, throughout the book.

The for Loop

A basic *for* loop has the same termination condition expression as the *while* loops, as well as two new sections: an *initialization block* and an *update statement*. [Figure 3.10](#) shows how these components are laid out.

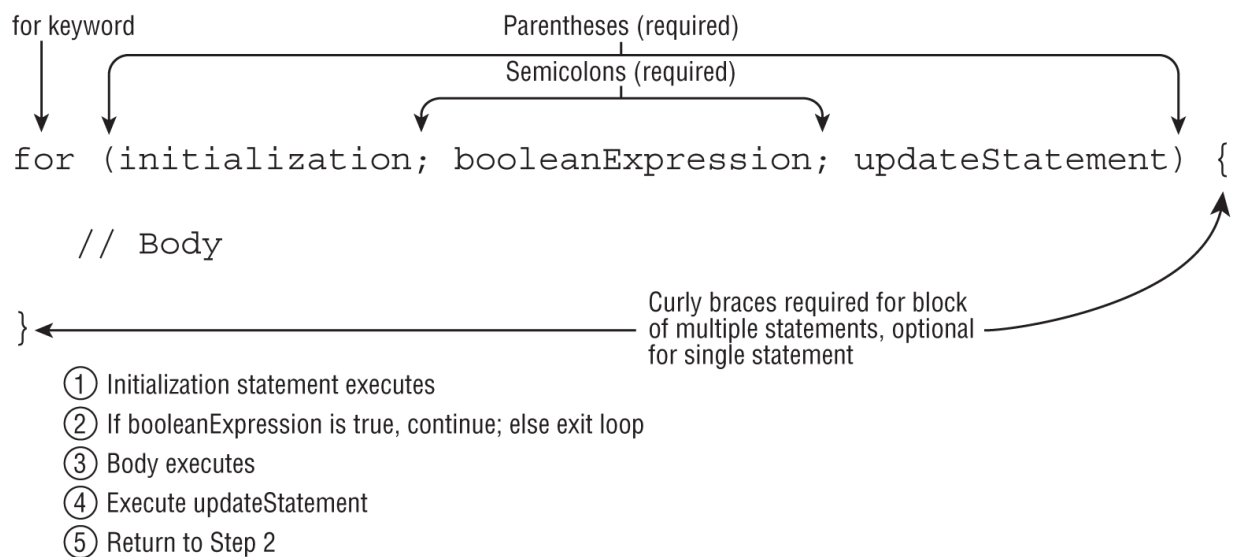


FIGURE 3.10 The structure of a basic for loop

Although [Figure 3.10](#) might seem a little confusing and almost arbitrary at first, the organization of the components and flow allow us to create extremely powerful statements in a single line that otherwise would take multiple lines with a *while* loop. Each of the three sections is separated by a semicolon. In addition, 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 accessible only within the for loop. Be wary of any exam questions in which a variable is declared within the initialization block of a for loop and then read outside the loop. For example, this code does not compile because the loop variable *i* is referenced outside the loop:

```
for (int i = 0; i < 10; i++)  
    System.out.println("Value is: "+i);  
System.out.println(i); // DOES NOT COMPILE
```

Alternatively, variables declared before the for loop and assigned a value in the initialization block may be used outside the for loop because their scope precedes the creation of the for loop.

```
int i;  
for (i = 0; i < 10; i++)  
    System.out.println("Value is: "+i);  
System.out.println(i);
```

Let's take a look at an example that prints the first five numbers, starting with zero:

```
for (int i = 0; i < 5; i++) {  
    System.out.print(i + " ");  
}
```

The local variable `i` is initialized first to 0. The variable `i` is only in scope for the duration of the loop and is not available outside the loop once the loop has completed. Like a `while` loop, the boolean condition is evaluated on every iteration of the loop *before* the loop executes. Since it returns true, the loop executes and outputs 0 followed by a space. Next, the loop executes the update section, which in this case increases the value of `i` to 1. The loop then evaluates the boolean expression a second time, and the process repeats multiple times, printing the following:

0 1 2 3 4

On the fifth iteration of the loop, the value of `i` reaches 4 and is incremented by 1 to reach 5. On the sixth iteration of the loop, the boolean expression is evaluated, and since `(5 < 5)` returns false, the loop terminates without executing the statement loop body.



Real World Scenario

Why *i* in *for* Loops?

You may notice it is common practice to name a *for* loop variable *i*. Long before Java existed, programmers started using *i* as short for increment variable, and the practice exists today, even though many of those programming languages no longer do! For double or triple loops, where *i* is already used, the next letters in the alphabet, *j* and *k*, are often used.

Printing Elements in Reverse

Let's say you wanted to print the same first five numbers from zero as we did in the previous section, but this time in reverse order. The goal then is to print 4 3 2 1 0.

How would you do that? An initial implementation might look like the following:

```
for (var counter = 5; counter > 0; counter--) {  
    System.out.print(counter + " ");  
}
```

While this snippet does output five distinct values, and it resembles our first *for* loop example, it does not output the same five values. Instead, this is the output:

```
5 4 3 2 1
```

Wait, that's not what we wanted! We wanted 4 3 2 1 0. It starts with 5, because that is the first value assigned to it. Let's fix that by starting with 4 instead:

```
for (var counter = 4; counter > 0; counter--) {  
    System.out.print(counter + " ");  
}
```

What does this print now? It prints the following:

4 3 2 1

So close! The problem is that it ends with 1, not 0, because we told it to exit as soon as the value was not strictly greater than 0. If we want to print the same 0 through 4 as our first example, we need to update the termination condition, like this:

```
for (var counter = 4; counter >= 0; counter--) {  
    System.out.print(counter + " ");  
}
```

Finally! We have code that now prints 4 3 2 1 0 and matches the reverse of our for loop example in the previous section. We could have instead used `counter > -1` as the loop termination condition in this example, although `counter >= 0` tends to be more readable.



For the exam, you are going to have to know how to read forward and backward for loops. When you see a for loop on the exam, pay close attention to the loop variable and operations if the decrement operator, `--`, is used. While incrementing from 0 in a for loop is often straightforward, decrementing tends to be less intuitive. In fact, if you do see a for loop with a decrement operator on the exam, you should assume they are trying to test your knowledge of loop operations.

Working with *for* Loops

Although most for loops you are likely to encounter in your professional development experience will be well defined and similar to the previous examples, there are a number of variations and edge cases you could see on the exam. You should familiarize yourself with the following five examples; variations of these are likely to be seen on the exam.

1. Creating an Infinite Loop


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

Although this for loop may look like it does not compile, it will in fact compile and run without issue. It is actually an infinite loop that will print the same statement repeatedly. This example reinforces the fact that the components of the for loop are each optional. Note that the semicolons separating the three sections are required, as for() without any semicolons will not compile.

2. 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 + " ");
}
```

This code demonstrates three variations of the for loop you may not have seen. First, you can declare a variable, such as x in this example, before the loop begins and use it after it completes. Second, your initialization block, boolean expression, and update statements can include extra variables that may or may not reference each other. For example, z is defined in the initialization block and is never used. Finally, the update statement can modify multiple variables. This code will print the following when executed:

```
0 1 2 3 4 5
```

3. Redeclaring a Variable in the Initialization Block

```
int x = 0;
for (int x = 4; x < 5; x++)    // DOES NOT COMPILE
    System.out.print(x + " ");
```

This example looks similar to the previous one, but it does not compile because of the initialization block. The difference is that the declaration of x is repeated in the initialization block after already being declared before the loop, resulting in the compiler stopping because of a duplicate variable declaration. We can fix this loop by removing the declaration of x from the for loop as follows:

```
int x = 0;
for ( ; x < 5; x++)
```

```
System.out.print(x + " ");
```

4. Using Incompatible Data Types in the Initialization Block

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

Like the third example, this code will not compile, although this time for a different reason. The variables in the initialization block must all be of the same type. In the multiple-terms example, y and z were both long, so the code compiled without issue; but in this example, they have different types, so the code will not compile.

5. 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
```

We covered this already at the start of this section, but it is so important for passing the exam that we discuss it again here. If you notice, x is defined in the initialization block of the loop and then used after the loop terminates. Since x was scoped only for the loop, using it outside the loop will cause a compiler error.

Modifying Loop Variables

As a general rule, it is considered a poor coding practice to modify loop variables as it can lead to an unexpected result, such as in the following examples:

```
for (int i = 0; i < 10; i++) // Infinite Loop  
    i = 0;  
  
for (int j = 1; j < 10; j++) // Iterates 5 times  
    j++;
```

It also tends to make code difficult for other people to follow.

The for-each Loop

The *for-each* loop is a specialized structure designed to iterate over arrays and various Collections Framework classes, as presented in [Figure 3.11](#).

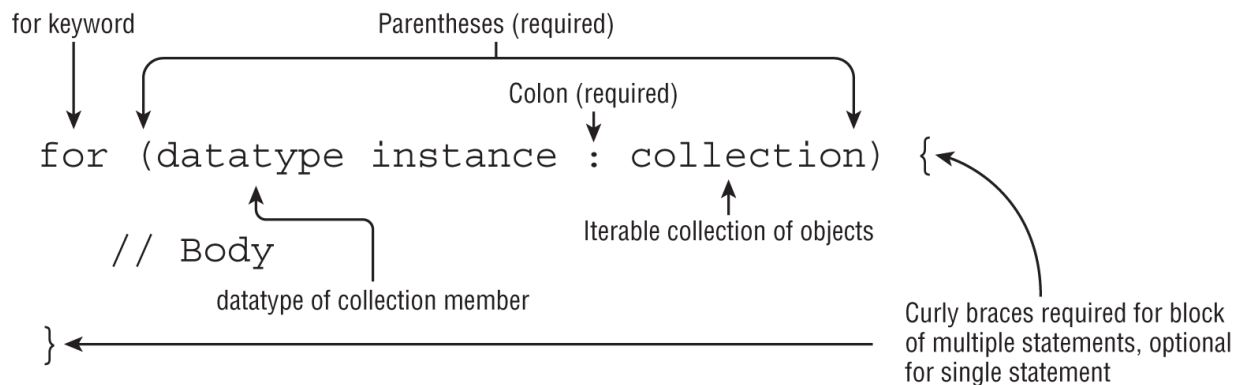


FIGURE 3.11 The structure of an enhanced for-each loop

The for-each loop declaration is composed of an initialization section and an object to be iterated over. The right side of the for-each loop must be one of the following:

- A built-in Java array
- An object whose type implements `java.lang.Iterable`

We cover what *implements* means in [Chapter 7](#), but for now you just need to know that the right side must be an array or collection of items, such as a `List` or a `Set`. For the exam, you should know that this does not include all of the Collections Framework classes or interfaces, but only those that implement or extend that `Collection` interface. For example, `Map` is not supported in a for-each loop, although `Map` does include methods that return `Collection` instances.

The left side of the for-each loop must include a declaration for an instance of a variable whose type is compatible with the type of the array or collection on the right side of the statement. On each iteration of the loop, the named variable on the left side of the statement is assigned a new value from the array or collection on the right side of the statement.

Compare these two methods that both print the values of an array, one using a traditional for loop and the other using a for-each loop:

```
void printNames(String[] names) {
    for (int counter = 0; counter < names.length; counter++)
        System.out.println(names[counter]);
}
```

```
void printNames(String[] names) {
    for (var name : names)
        System.out.println(name);
}
```

The for-each loop is a lot shorter, isn't it? We no longer have a counter loop variable that we need to create, increment, and monitor. Like using a for loop in place of a while loop, for-each loops are meant to reduce boilerplate code, making code easier to read/write, and freeing you to focus on the parts of your code that really matter.

We can also use a for-each loop on a List, since it implements Iterable.

```
void printNames(List<String> names) {
    for (var name : names)
        System.out.println(name);
}
```

We cover generics in detail in [Chapter 9](#), “Collections and Generics.” For this chapter, you just need to know that on each iteration, a for-each loop assigns a variable with the same type as the generic argument. In this case, name is of type String.

So far, so good. What about the following examples?

```
String birds = "Jay";
for (String bird : birds) // DOES NOT COMPILE
    System.out.print(bird + " ");

String[] sloths = new String[3];
for (int sloth : sloths) // DOES NOT COMPILE
    System.out.print(sloth + " ");
```

The first for-each loop does not compile because the String birds cannot be used on the right side of the statement. While a String may represent a list of characters, it has to actually be an array or implement Iterable. The second example does not compile because the loop variable type on the left side of the statement is int and doesn't match the expected type of String.

Controlling Flow with Branching

The final types of control flow structures we cover in this chapter are branching statements. Up to now, we have been dealing with single loops that ended only when their boolean expression evaluated to false. We now show you other ways loops could end, or branch, and you see that the path taken during runtime may not be as straightforward as in the previous examples.

Nested Loops

Before we move into branching statements, we need to introduce the concept of nested loops. A *nested loop* is a loop that contains another loop, including while, do/while, for, and for-each loops. For example, consider the following code that iterates over a two-dimensional array, which is an array that contains other arrays as its members. We cover arrays in detail in [Chapter 4](#), “Core APIs,” but for now, assume the following is how you would declare an array of arrays:

```
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();
}
```

Notice that we intentionally mix a for loop and a for-each loop in this example. The outer loop will execute a total of three times. Each time the outer loop executes, the inner loop is executed four times. When we execute this code, we see the following output:

5	2	1	3
3	9	8	9
5	7	12	7

Nested loops can include while and do/while, as shown in this example. See whether you can determine what this code will output:

```
int hungryHippopotamus = 8;
while (hungryHippopotamus > 0) {
```

```

    do {
        hungryHippopotamus -= 2;
    } while (hungryHippopotamus>5);
    hungryHippopotamus--;
    System.out.print(hungryHippopotamus+", ");
}

```

The first time this loop executes, the inner loop repeats until the value of hungryHippopotamus is 4. The value will then be decremented to 3, and that will be the output at the end of the first iteration of the outer loop.

On the second iteration of the outer loop, the inner do/while will be executed once, even though hungryHippopotamus is already not greater than 5. As you may recall, do/while statements always execute the body at least once. This will reduce the value to 1, which will be further lowered by the decrement operator in the outer loop to 0. Once the value reaches 0, the outer loop will terminate. The result is that the code will output the following:

3, 0,

The examples in the rest of this section include many nested loops. You will also encounter nested loops on the exam, so the more practice you have with them, the more prepared you will be.

Adding Optional Labels

One thing we intentionally skipped when we presented if statements, switch statements, and loops is that they can all have optional labels. 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 identifier that is followed by a colon (:). For example, we can add optional labels to one of the previous examples:

```

int[][] myComplexArray = {{5,2,1,3}, {3,9,8,9}, {5,7,12,7}};

OUTER_LOOP: for (int[] mySimpleArray : myComplexArray) {
    INNER_LOOP: for (int i = 0; i < mySimpleArray.length; i++)
    {
        System.out.print(mySimpleArray[i]+"\\t");
    }
    System.out.println();
}

```

Labels follow the same rules for formatting as identifiers. For readability, we show them in *snake_case*, with uppercase letters and underscores between words. When dealing with only one loop, labels do not add any value, but as you learn in the next section, they are extremely useful in nested structures.



While this topic is not on the exam, it is possible to add optional labels to control and block statements. For example, the following is permitted by the compiler, albeit extremely uncommon:

```
int frog = 15;  
BAD_IDEA: if (frog > 10)  
EVEN_WORSE_IDEA: {  
    frog++;  
}
```

The *break* Statement

As you saw when working with switch statements, a *break* statement transfers the flow of control out to the enclosing statement. The same holds true for a break statement that appears inside of a while, do/while, or for loop, as it will end the loop early, as shown in [Figure 3.12](#).

Optional reference to head of loop

Colon (required if optionalLabel is present)

```
optionalLabel: while (booleanExpression) {

    // Body

    // Somewhere in the loop
    break optionalLabel;
}
```

break keyword

Semicolon (required)

FIGURE 3.12 The structure of a break statement

Notice in [Figure 3.12](#) that 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. In the following example, we search for the first (x, y) array index position of a number within an unsorted two-dimensional array:

```
10: public class FindInMatrix {
11:     public static void main(String[] args) {
12:         int[][] list = {{1,13}, {5,2}, {2,2}};
13:         int searchValue = 2;
14:         int positionX = -1;
15:         int positionY = -1;
16:
17:         PARENT_LOOP: for (int i = 0; i < list.length; i++) {
18:             for (int j = 0; j < list[i].length; j++) {
19:                 if (list[i][j] == searchValue) {
20:                     positionX = i;
21:                     positionY = j;
22:                     break PARENT_LOOP;
23:                 }
24:             }
25:         }
26:         if (positionX == -1 || positionY == -1) {
27:             System.out.print("Value "+searchValue+" not
found");
28:         } else {
```



```

29:          System.out.print("Value "+searchValue+" found at:
" +
30:          "("+positionX+", "+positionY+")");
31:      }
32:  } }

```

When executed, this code will output the following:

Value 2 found at: (1,1)

In particular, take a look at the statement `break PARENT_LOOP`. This statement will break out of the entire loop structure as soon as the first matching value is found. Now, imagine what would happen if we replaced the body of the inner loop with the following:

```

19:          if (list[i][j]==searchValue) {
20:              positionX = i;
21:              positionY = j;
22:              break;
23:          }

```

How would this change our flow, and would the output change? Instead of exiting when the first matching value is found, the program would now only exit the inner loop when the condition was met. In other words, the structure would find the first matching value of the last inner loop to contain the value, resulting in the following output:

Value 2 found at: (2,0)

Finally, what if we removed the break altogether?

```

19:          if (list[i][j]==searchValue) {
20:              positionX = i;
21:              positionY = j;
22:          }
23:      }

```

In this case, the code would search for the last value in the entire structure that had the matching value. The output would look like this:

Value 2 found at: (2,1)

You can see from this example that using a label on a break statement in a nested loop, or not using the break statement at all, can cause the loop

structure to behave quite differently.

The *continue* Statement

Let's now extend our discussion of advanced loop control with the *continue* statement, a statement that causes flow to finish the execution of the current loop iteration, as shown in [Figure 3.13](#).

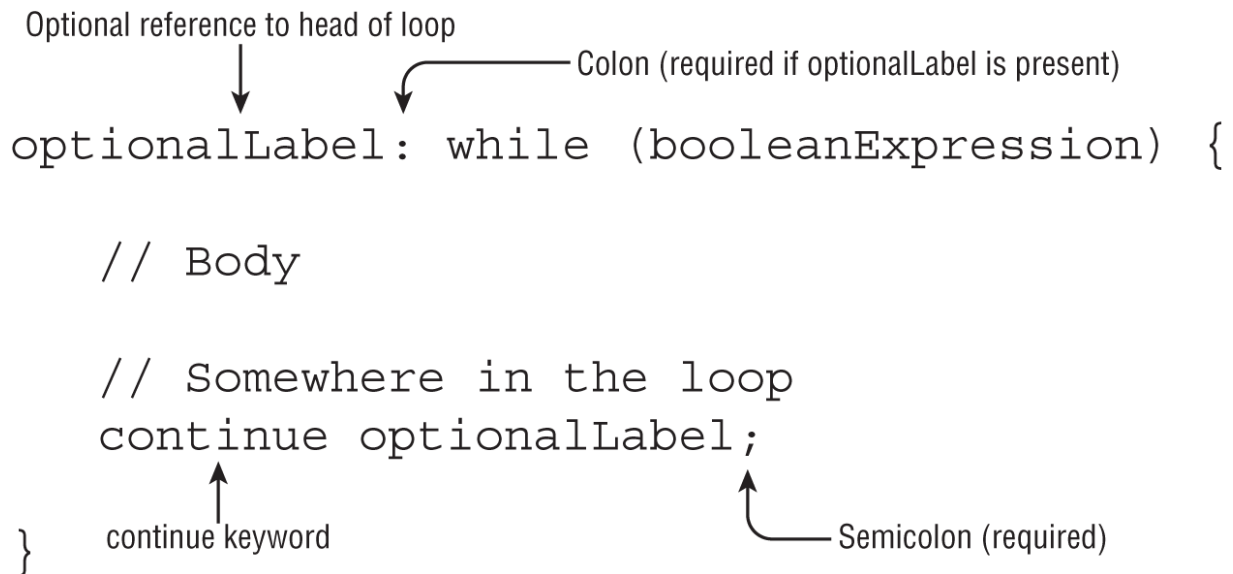


FIGURE 3.13 The structure of a *continue* statement

You may notice that the syntax of the *continue* statement mirrors that of the *break* statement. In fact, the statements are identical in how they are used, but with different results. While 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.

Let's take a look at an example. Imagine we have a zookeeper who is supposed to clean the first leopard in each of four stables but skip stable b entirely.

```
1: public class CleaningSchedule {  
2:     public static void main(String[] args) {  
3:         CLEANING: for (char stables = 'a'; stables<='d';  
stables++) {
```

```

4:         for (int leopard = 1; leopard <= 3; leopard++) {
5:             if (stables=='b' || leopard==2) {
6:                 continue CLEANING;
7:             }
8:             System.out.println("Cleaning:
"+stables+", "+leopard);
9: } } } }

```

With the structure as defined, the loop will return control to the parent loop any time the first value is b or the second value is 2. On the first, third, and fourth executions of the outer loop, the inner loop prints a statement exactly once and then exits on the next inner loop when leopard is 2. On the second execution of the outer loop, the inner loop immediately exits without printing anything since b is encountered right away. The following is printed:

```

Cleaning: a,1
Cleaning: c,1
Cleaning: d,1

```

Now, imagine we remove the CLEANING label in the continue statement so that control is returned to the inner loop instead of the outer. Line 6 becomes the following:

```

6:                 continue;

```

This corresponds to the zookeeper cleaning all leopards except those labeled 2 or in stable b. The output is then the following:

```

Cleaning: a,1
Cleaning: a,3
Cleaning: c,1
Cleaning: c,3
Cleaning: d,1
Cleaning: d,3

```

Finally, if we remove the continue statement and the associated if statement altogether by removing lines 5–7, we arrive at a structure that outputs all the values, such as this:

```

Cleaning: a,1
Cleaning: a,2
Cleaning: a,3
Cleaning: b,1

```

Cleaning: b,2
Cleaning: b,3
Cleaning: c,1
Cleaning: c,2
Cleaning: c,3
Cleaning: d,1
Cleaning: d,2
Cleaning: d,3

The *return* Statement

Given that this book shouldn't be your first foray into programming, we hope you've come across methods that contain return statements.

Regardless, we cover how to design and create methods that use them in detail in [Chapter 5](#).

For now, though, you should be familiar with the idea that creating methods and using return statements can be used as an alternative to using labels and break statements. For example, take a look at this rewrite of our earlier FindInMatrix class:

```
public class FindInMatrixUsingReturn {
    private static int[] searchForValue(int[][] list, int v) {
        for (int i = 0; i < list.length; i++) {
            for (int j = 0; j < list[i].length; j++) {
                if (list[i][j] == v) {
                    return new int[] {i, j};
                }
            }
        }
        return null;
    }

    public static void main(String[] args) {
        int[][] list = { { 1, 13 }, { 5, 2 }, { 2, 2 } };
        int searchValue = 2;
        int[] results = searchForValue(list, searchValue);

        if (results == null) {
            System.out.print("Value " + searchValue + " not
found");
        } else {
            System.out.print("Value " + searchValue + " found at:
" +
                "(" + results[0] + ", " + results[1] + ")");
        }
    }
}
```

```
}  
}
```

This class is functionally the same as the first `FindInMatrix` class we saw earlier using `break`. If you need finer-grained control of the loop with multiple `break` and `continue` statements, the first class is probably better. That said, we find code without labels and `break` statements a lot easier to read and debug. Also, making the search logic an independent function makes the code more reusable and the calling `main()` method a lot easier to read.

For the exam, you will need to know both forms. Just remember that `return` statements can be used to exit loops quickly and can lead to more readable code in practice, especially when used with nested loops.

Unreachable Code

One facet of `break`, `continue`, and `return` that you should be aware of is that any code placed immediately after them in the same block is considered unreachable and will not compile. For example, the following code snippet does not compile:

```
int checkDate = 0;  
while (checkDate<10) {  
    checkDate++;  
    if (checkDate>100) {  
        break;  
        checkDate++; // DOES NOT COMPILE  
    }  
}
```

Even though it is not logically possible for the `if` statement to evaluate to true in this code sample, the compiler notices that you have statements immediately following the `break` and will fail to compile with “unreachable code” as the reason. The same is true for `continue` and `return` statements, as shown in the following two examples:

```
int minute = 1;  
WATCH: while (minute>2) {  
    if (minute++>2) {  
        continue WATCH;  
        System.out.print(minute); // DOES NOT COMPILE  
    }  
}
```

```

}

int hour = 2;
switch (hour) {
    case 1: return; hour++; // DOES NOT COMPILE
    case 2:
}

```

One thing to remember is that it does not matter if the loop or decision structure actually visits the line of code. For example, the loop could execute zero or infinite times at runtime. Regardless of execution, the compiler will report an error if it finds any code it deems unreachable, in this case any statements immediately following a break, continue, or return statement.

Reviewing Branching

We conclude this section with [Table 3.1](#), which will help remind you when labels and other various statements are permitted in Java. For illustrative purposes our examples used these statements in nested loops, although they can be used inside single loops as well.

[TABLE 3.1](#) Supported control statement features

	Labels	break	continue	yield	when
while	Yes	Yes	Yes	No	No
do/while	Yes	Yes	Yes	No	No
for	Yes	Yes	Yes	No	No
switch	Yes	Yes	No	Yes	Yes



Some of the most time-consuming questions on the exam could involve nested loops with lots of branching. Unless you can spot a compiler error right away, you might consider skipping these questions and coming back to them at the end. Remember, all questions on the exam are weighted evenly!

Summary

This chapter presented how to make intelligent decisions in Java. We covered basic decision-making constructs such as `if`, `else`, and `switch` and showed how to use them to change the path of the process at runtime. We also covered `switch` expressions and showed how they often lead to more concise code.

In both the `if` and `switch` sections, we showed how to apply pattern matching to reduce boilerplate code. Pattern matching, especially with `switch`, is one of the newer features of Java 21, so expect to see at least one question on the exam on it.

We then moved our discussion to repetition control structures, otherwise known as loops. We showed how to use `while` and `do/while` loops to create processes that execute multiple times and also showed how it is important to make sure they eventually terminate. Remember that most of these structures require the evaluation of the termination condition, represented as a boolean expression, to complete.

Next, we covered the extremely convenient repetition control structures: the `for` and `for-each` loops. While their syntax is more complex than the traditional `while` or `do/while` loops, they are extremely useful in everyday coding and allow you to create complex expressions in a single line of code. With a `for-each` loop, you don't need to explicitly write a boolean expression, since the compiler builds one for you.

We concluded this chapter by discussing advanced control options and how flow can be enhanced through nested loops coupled with `break`, `continue`, and `return` statements. Be wary of questions on the exam that use nested loops, especially ones with labels, and verify that they are being used correctly.

This chapter is especially important because at least one component of this chapter will likely appear in every exam question with sample code. Many of the questions on the exam focus on proper syntactic use of the structures, as they will be a large source of questions that end in “Does not compile.” You should be able to answer all of the review questions correctly or fully understand those that you answered incorrectly before moving on to later chapters.

Exam Essentials

Understand *if* and *else* decision control statements. The `if` and `else` statements come up frequently throughout the exam in questions unrelated to decision control, so make sure you fully understand these basic building blocks of Java.

Apply pattern matching and flow scoping to *if*. Pattern matching can be used to reduce boilerplate code of some `if` statements, by applying the `instanceof` operator and a variable type/name. It can also include a guard, which is an optional conditional clause, after the pattern variable declaration. Pattern matching uses flow scoping in which the pattern variable is in scope as long as the compiler can definitively determine its type.

Understand *switch* statements and their proper usage. You should be able to spot a poorly formed `switch` statement on the exam. The `switch` value and data type should be compatible with the case clauses, and the values for the case clauses must evaluate to compile time constants. Finally, at runtime, a `switch` statement branches to the first matching case, or `default` if there is no match, or exits entirely if there is no match and no `default` branch. The process then continues into any proceeding case or `default` clause until a `break` or `return` statement is reached.

Use *switch* expressions correctly. Discern the differences between switch statements and switch expressions. Understand how to write switch expressions correctly, including proper use of semicolons, writing case expressions and blocks that yield a consistent value, and making sure all possible values of the switch variable are handled by the switch expression.

Apply pattern matching to *switch*. Understand how switch statements and expressions support pattern matching and allow any object to be used as the switch variable. It also supports a case branch with a guard, via the `when` keyword. Pattern matching alters two common rules with switch: a switch statement now must be exhaustive when pattern matching is used, and the ordering of switch expression branches is now important.

Write *while* loops. Know the syntactical structure of all `while` and `do/while` loops. In particular, know when to use one versus the other.

Be able to use *for* loops. You should be familiar with `for` and `for-each` loops and know how to write and evaluate them. Each loop has its own special properties and structures. You should know how to use `for-each` loops to iterate over lists and arrays.

Understand how *break*, *continue*, and *return* can change flow control.

Know how to change the flow control within a statement by applying a `break`, `continue`, or `return` statement. Also know which control statements can accept `break` statements and which can accept `continue` statements. Finally, you should understand how these statements work inside embedded loops or switch statements.

Review Questions

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

1. What is the output of the following code snippet?

```
32: Object skips = 10;
33: switch (skips) {
34:     case a when a < 10 -> System.out.print(2);
35:     case b when b >= 10 -> System.out.print(4);
36:     case null -> System.out.print(6);
```