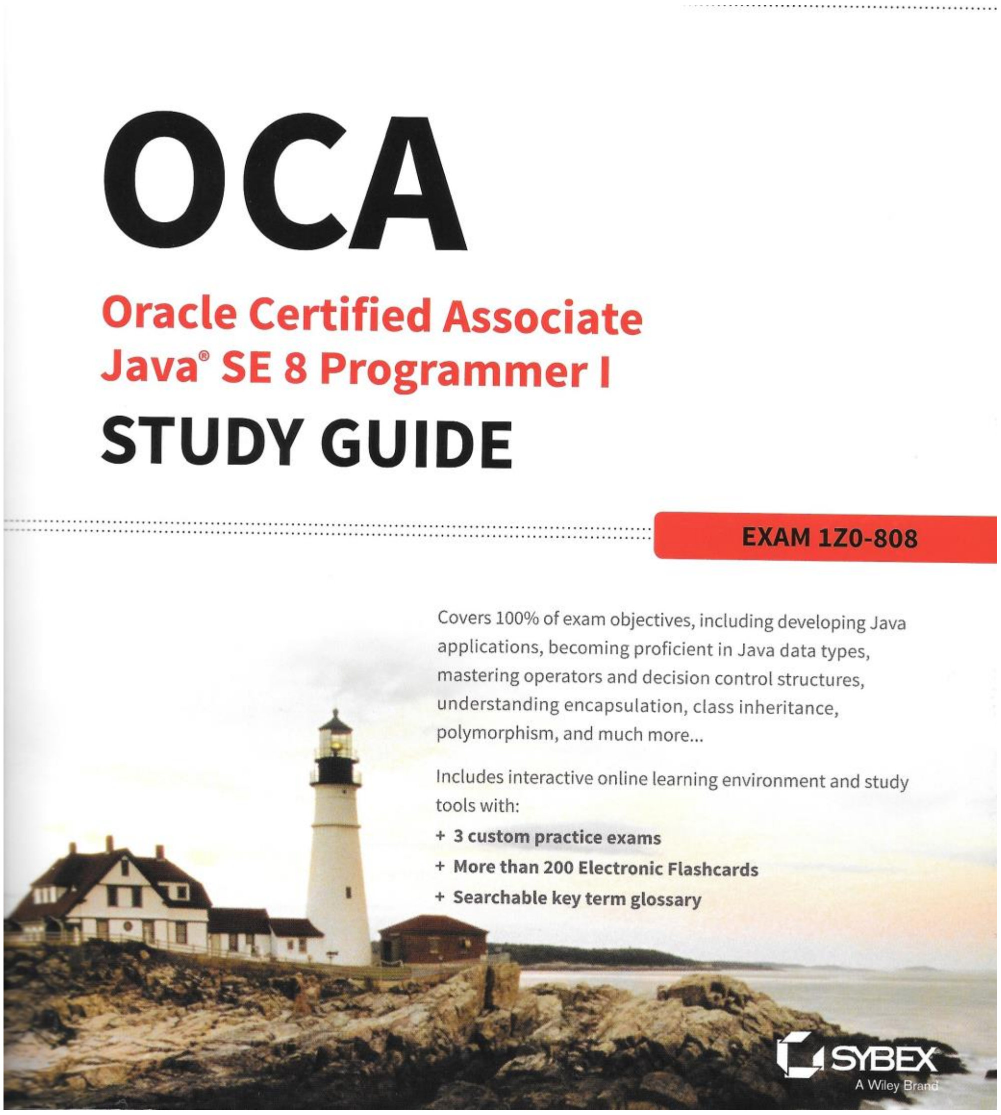
OCA

Oracle Certified Associate

Java SE 8 Programmer I

STUDY GUIDE

SUMMARY



OCA
Oracle Certified Associate
Java® SE 8 Programmer I
STUDY GUIDE

EXAM 1Z0-808

Covers 100% of exam objectives, including developing Java applications, becoming proficient in Java data types, mastering operators and decision control structures, understanding encapsulation, class inheritance, polymorphism, and much more...

Includes interactive online learning environment and study tools with:

+ 3 custom practice exams
+ More than 200 Electronic Flashcards
+ Searchable key term glossary

SYBEX
A Wiley Brand

Table of Contents

Introduction 6

1 Java Building Blocks 7

1.1 OCA Exam objectives 7

1.2 Understanding the Java Class Structure 7

1.2.1 Fields and Methods 7

1.2.2 Comments 7

1.2.3 Classes vs. Files 8

1.3 Writing a main() Method 8

1.4 Understanding Package Declarations and Imports 9

1.4.1 Wildcards 9

1.4.2 Redundant imports 9

1.4.3 Naming Conflicts 10

1.4.4 Creating a New Package 10

1.4.5 Code Formatting on the Exam 10

1.5 Creating Objects 10

1.5.1 Constructors 10

1.5.2 Reading and Writing Object Fields 10

1.5.3 Instance Initializer Blocks 11

1.6 Distinguishing Between Object References and Primitives 11

1.6.1 Primitive Types 11

1.6.2 Reference Types 12

1.6.3 Key Differences Between Primitives & Reference Types 12

1.7 Declaring and Initializing Variables 12

1.7.1 Declaring Multiple Variables 12

1.7.2 Identifiers 12

1.8 Understanding Default Initialization of Variables 13

1.8.1 Local Variables 13

1.8.2 Instance and Class Variables 13

1.9 Understanding Variable Scope 13

1.10 Ordering Elements in a Class 14

1.11 Destroying Objects 14

1.11.1 Garbage Collection 14

1.11.2 finalize() 14

1.12 Benefits of Java 15

1.13 Exam Essentials 15

2 Operators and Statements 16
2.1 OCA Exam Objectives 16
2.2 Understanding Java Operators 16
2.3 Working with Binary Arithmetic Operators 16
2.3.1 Arithmetic Operators 16
2.3.2 Numeric Promotion 17
2.4 Working with Unary Operators 17
2.4.1 Logical Complement and Negation Operators 17
2.5 Increment and Decrement Operators 18
2.6 Using Additional Binary Operators 18
2.6.1 Casting Primitive Values 18
2.6.2 Compound Assignment Operators 18
2.6.3 Relational Operators 19
2.6.4 Logical Operators 19
2.7 Understanding Java Statements 20
2.7.1 The if-then Statement 20
2.7.2 The if-then-else Statement 20
2.7.3 Ternary Operator 21
2.7.4 The switch Statement 22
2.7.5 The while Statement 23
2.7.6 The do-while Statement 24
2.7.7 The for Statement 24
2.8 Understanding Advanced Flow Control 26
2.8.1 Nested Loops 26
2.8.2 Adding Optional Labels 26
2.8.3 The break Statement 26
2.8.4 The continue Statement 26
2.9 Exam Essentials 27
Appendix B – Study Tips 28
Taking the test 28
Reviewing Common Compiler Issues 28

Introduction

- URL blog: www.selikoff.net/oca → check for updates about changes on topics on the exam!
- Read Appendix B first.

1 Java Building Blocks

1.1 OCA Exam objectives

1. Java Basics:
    a. Define the scope of variables.
    b. Define the structure of a Java class.
    c. Create executable Java applications with a main method; run a Java program from the command line; including console output.
    d. Import other Java packages to make them accessible in your code.
    e. Compare and contrast the features and components of Java such as platform independence, object orientation, encapsulation, etc.
2. Working with Java Data Types:
    a. Declare and initialize variables (including casting or primitive types).
    b. Differentiate between object reference variables and primitive variables.
    c. Know how to read or write to object fields.
    d. Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection).

1.2 Understanding the Java Class Structure

- Classes are basic building blocks.
- A class describes all parts and characteristics of one of those building blocks.
- To use them you create objects.
- An object is a runtime instance of a class in memory.
- All various objects of all different classes represent the state of your program.

1.2.1 Fields and Methods

- Classes can have the following members:
    - fields (variables): hold the state of the program
    - methods (functions or procedures): operate on the state
- Keyword: a word with special meaning, like `public` and `class`, meaning a public class can be used by other classes.
- Example of a class with a field and methods:

```
public class Animal {          // defines a public class named Animal which can be used by other classes

   String name;                // define a variable called name of type String

   public String getName() {   // define public method that may be called from other classes
      return name;             // with return type String
   }

   public void setName(String newname) {  // define public method which requires information to be
      name = newname;                     // supplied to it from calling method (parameter: newname
   }                                      // of type String and returns nothing (type void))
}
```

- The full declaration of a method is called a method signature.

1.2.2 Comments

- Comments aren't executable code, you can place them anywhere.
- Three types of comments:
    - Single-line comment:

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

- Multiple-line comment:

```
/* Multiple
 * line comment (the * here is optional, but added for readability)
 */
```

- Javadoc comment:

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

1.2.3 Classes vs. Files

- Most of the time each Java class is defined in its own `*.java` file which is usually `public`, meaning any code can call it.
- You can put two classes in the same file, but at most one of the classes is then allowed to be `public`.
- If you have a `public class`, it needs to match the filename (`public class Animal2` would not compile in a file named `Animal.java`, but it would if the file is named `Animal2.java`).

1.3 Writing a *main()* Method

- A Java program begins execution with its `main()` method.
- Comments aren't executable code, you can place them anywhere.
- The signature is:

```
public class Zoo { // we need a class structure to start a Java program because the language requires it
   public static void main(String[] args) {     // this declares the entry point
   }
}
```

- To compile the source code we do: `javac Zoo.java`.
  - To compile it, it needs to have the extension `.java` and match the name of the class (including case).
  - The result is a file of bytecode with the same name but with the `.class` extension: `Zoo.class`.
  - Bytecode consists of instructions that the JVM knows how to execute.
- To execute the code we do: `java Zoo`.
  - We leave out the `.class` extensions on execution, since the period has a reserved meaning in the JVM.
- Main method signature:
  - ***public***
    - Access modifier → declares method's level of exposure to potential callers in the program; public means anywhere.
  - ***static***
    - Binds a method to its class so it can be called by just the class name, e.g. `Zoo.main()`. Java doesn't need to create an object to call the `main()` method.
    - If a `main()` method isn't present in the class we name with the `.java` executable, the process will throw an error and terminate. Even if a `main()` method is present, Java will throw an exception if it isn't static. A nonstatic `main()` method might as well be invisible from the point of view of the JVM.
  - ***void***
    - Represents the return type. In general it's good practice to use void for methods that change an object's state.
  - ***(String[] args)***
    - Represents the parameter list.
    - Can also be written as `String args[]` or `String… args`;
    - `args` hints this list contains values that were read in (arguments) when the JVM started (any name can be used though).
    - `[]` represents an array, a fixed-size list of items all of the same type.

        - … represents varargs (variable argument lists).
        - Array indexes begin with 0.
        - Spaces are used to separate arguments on the command line, e.g. `java Zoo Bronx Zoo`:
            - `args[0]` → Bronx
            - `args[1]` → Zoo
        - For spaces inside an argument, surround them in quotes, e.g. `java Zoo “San Diego” Zoo`:
            - `args[0]` → San Diego
            - `args[1]` → Zoo
        - All command-line arguments are treated as String objects.
        - If you don’t pass enough arguments and try to access it, Java prints out an exception, e.g. running `java Zoo Bronx` and then trying to access `args[1]` results in `java.lang.ArrayIndexOutOfBoundsException`.
- You need a JDK to compile because it includes a compiler.
- To run already compiled code, a JRE is enough.
- Java class files run on the JVM and therefore run on any machine with Java.

1.4 Understanding Package Declarations and Imports

- Java puts classes in packages, logical groupings for classes.
- In Java you need to tell it in which packages to look in to find code.
- An error like `Random cannot be resolved to a type` can mean two things:
    - A typo in the name of the class.
    - Omitting a needed import statement.
- Import statements tell Java in which packages to look in for classes.
- Java only looks for class names in the packages.
- Package names are hierarchical: start reading package name at the beginning.
- If package name begins with `java` or `javax`, it comes with the JDK.
- Detailed packages are called child packages: `com.amazon.java8book` is a child package of `com.amazon` (it’s longer thus more specific).
- The rule for package names is that they are mostly letters or numbers separated by dots (the same as for variable names).
- The exam doesn’t try to trick you by giving invalid package names, but it can trick you giving invalid variable names though.

1.4.1 Wildcards

- Classes in the same package are often imported together, e.g. `import java util.*;`
- The * is a wildcard that matches all classes in the package. It doesn’t import child packages, fields, or methods; it imports only classes.
- Including so many classes doesn’t slow down you program; the compiler figures out what’s actually needed.

1.4.2 Redundant imports

- The package `java.lang` is automatically imported.
- You also don’t have to import classes that are in the same package as the class importing it. Java automatically looks in the current package for other classes.
- Examples given that classes `Files` and `Paths` are located in `java.nio.file`:

```
import java.nio.file.*;       // CORRECT: import both at the same time
import java.nio.file.Files;   // CORRECT: import Files explicitly
import java.nio.file.Paths;   // CORRECT: import Paths explicitly
import java.nio.*;            // INCORRECT: * only matches class names, not subpackages as “file.*Files”
import java.nio.*.*;          // INCORRECT: only one wildcard allowed and must be at the end
import java.nio.file.Paths.*; // INCORRECT: cannot import methods, only class names
```

1.4.3 Naming Conflicts

- One of the reasons for using packages is so that names don't have to be unique across all of Java, e.g. you can have two different `Date` classes in different packages: `java.util.Date` and `java.sql.Date`.
- It can be tricky though when you have multiple imports like:

```
import java.util*;
import java.sql.*;           // does not compile → the type Date is ambiguous
```

But this will work:

```
import java.util.Date;       // explicitly imported class name has precedence over any wildcards present
import java.sql.*;
```

But with "ties"of precedence:

```
import java.util.Date;       // this will fail, there can't be two defaults; results in:
import java.sql.Date;        // The import java.sql.Date collides with another import statement
```

- If you need both classes, you can pick one to use in the import and for the other you use the fully qualified class name. Or you could leave out any import and for both used the fully qualified class name.

1.4.4 Creating a New Package

- If no package name is used in a class, it has the default package scope. This should not be used, only for throwaway code. In real life, always name your packages to avoid naming conflicts and to allow others to reuse your code.
- The directory structure on your computer is related to the package name.

1.4.5 Code Formatting on the Exam

- If the exam isn't asking about imports in the question, it will often omit the imports to save space. In that case you'll see examples with line numbers that don't begin with 1.
- When you do see the line number 1 or no line numbers at all, you have to make sure imports aren't missing.
- You'll also see code merged on the same line.
- You'll also see code that doesn't have a `main()` method. When this happens, assume the `main()` method, class definition , and all necessary imports are present.

1.5 Creating Objects

1.5.1 Constructors

- To create an instance of a class, you write `new` before it, e.g. `Random r = new Random();`
  - First you declare the type you'll be creating and then give the variable a name, i.e. a place to store a reference to the object.
  - Then you write `new Random()` to actually create the object.
- `Random()` is a constructor, a special type of method that creates a new object.
- Two key points of a constructor:
  - The name of the constructor matches the name of the class.
  - There is no return type.
- When you see a method name beginning with a capital letter and having a return type, pay special attention to it. It is not a constructor since there's a return type. It's a regular method.
- Purpose of constructor: initialize fields.
- Fields can also be initialized directly on the line on which they're declared, e.g. `int numEggs = 0;`
- If no constructor is defined, the compiler will supply a "do nothing" default constructor.

1.5.2 Reading and Writing Object Fields

- Reading a variable is known as getting it (getter).
- Writing to a variable is known as setting it (setter).

- You can read and write instance variables directly from the caller (depending on the access modifier used), e.g. `mother.numberEggs = 1` sets variable and `mother.numberEggs` reads variable.
- Fields can be read and written to directly on the line declaring them, e.g.

  ```
  String first = Theodore"; String last = "Moose"; String full = first + last;
  ```

  first and last are written to and full reads first and last and writes to full.

1.5.3 Instance Initializer Blocks

- The code between braces – `{}` – is called a code block.
- Blocks can be inside a method and are run when the method is called.
- Blocks can be outside a method, called instance initializers.

1.5.3.1 Order of Initialization

- 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
- You can't refer to a variable before it has been initialized.

1.6 Distinguishing Between Object References and Primitives

1.6.1 Primitive Types

- Java has eight built-in data types (primitive types).
- These represent the building blocks for Java objects; all Java objects are just a complex collection of these primitive data types.
- Java primitive types:

Keyword	Type	Example	Description
`boolean`	`true` or `false`	`true`	
`byte`	8-bit integral value	123	used for numbers without decimal points
`short`	16-bit integral value	123	used for numbers without decimal points
`int`	32-bit integral value	123	used for numbers without decimal points
`long`	64-bit integral value	123	used for numbers without decimal points
`float`	32-bit floating-point value	123.45f	used for floating-point (decimal) values requires letter f following the number
`double`	64-bit floating-point value	123.456	used for floating-point (decimal) values
`char`	16-bit Unicode value	'a'	

- Each numeric type uses twice as many bits as the smaller similar type.
- You should know that a byte can hold a value from –128 to 127.
- The number of bits is used by Java when it figures out how much memory to reserve for your variable, e.g. Java allocates 32 bits if you write `int num;`
- When a number is present in the code, it is called a literal.
- The following does not compile: `long max = 3123456789;` the value is seen as an `int` which can't be that large. To make it a long we have to do: `long max = 3123456789L;` by adding the `L` Java knows it's a long.
- Java allows you to specify digits in several other formats than base 10 (0 – 9):
  - Octal (digits 0–7), uses 0 as prefix, e.g. 017 which is base 8, so rightmost digit 7 is worth 7 and 2nd to rightmost digit is 1 is worth 8 (1 * 8), so in base 10: 7 + 8 = 15.
  - Hexadecimal (digits 0–9 and letters A–F), e.g. 0x1F which is base 16, so rightmost F is worth 15 and 2nd rightmost digit is 1 is worth 16 (1 * 16), so in base 10: 15 + 16 = 31.
  - Binary (digits 0–1), uses number 0 followed by a b or B as prefix, e.g. 0b11 which is base 2, so rightmost 1 is worth 1 and 2nd to rightmost digit 1 is worth 2 (1 * 2), so in base 10: 1 + 2 = 3.
- You won't need to convert between number systems on the exam though. You'll have to recognize valid literal values that can be assigned to numbers.

- Since Java 7 you can have underscores in numbers to make them easier to read as long as they are directly between two other numbers (so not at the beginning or the end of a literal and not right before or after a decimal point).

1.6.2 Reference Types

- A reference type refers to an object (instance of a class).
- Primitive types hold their values in memory where the variable is allocated.
- A reference "points" to an object by storing the memory address where the object is located (pointer).
- You can only use the reference to refer to the object.
- Examples:

```
java.util.Date date;  // date is a reference of type Date and can only point to a Date object
java.util.Date today; // today is also a reference of type Date and can only point to a Date object;
String greeting;      // greeting is a reference that can only point to a String object.
```

- A value is assigned to a reference in one of two ways:
  - A reference can be assigned to another object of the same type, e.g. `today = date();`
  - A reference can be assigned to a new object using the `new` keyword, e.g. `today = new java.util.Date();` or `greeting = "How are you?";`

1.6.3 Key Differences Between Primitives & Reference Types

- Reference types can be assigned `null`, meaning they do not currently refer to an object, primitives cannot be `null`; assigning `null` to a primitive will result in a compiler error.
- Reference types can be used to call methods when they do not point to `null`. Primitive types do not have methods declared on them.
- All primitive types have lower case type names. All classes that come with Java begin with uppercase.

1.7 Declaring and Initializing Variables

- A variable is a name for a piece of memory that stores data.
- When declaring a variable you state the variable type along with giving it a name, e.g. `String zooName;`
- Giving a variable a value is called initializing, which is done by using the equal sign after the variable name followed by the desired value, e.g. `zooName = "The Best Zoo";` or by doing it in the same statement as the declaration, e.g. `String zooName = "The Best Zoo";`

1.7.1 Declaring Multiple Variables

- You can declare many variables in the same declaration as long as they are all of the same type.
- You can also initialize any or all of those values inline.
- Examples:

```
String s1, s2;                // declared variables but not yet initialized

String s3 = "yes", s4 = "no"; // declared variables and initialized

int i1, i2, i3 = 0;           // tricky: only i3 is initialized, i1 and i2 are not yet initialized
                              // each snippet separated by a comma is a little declaration of its own

int num, String value;        // does not compile, tries to declare multiple variables of different
                              // types in the same statement; only works if they share the same type

double d1, double d2;         // not legal: not allowed to declare two different types in the same
                              // statement. Aren't they the same type (double)? Yes, but if you want
                              // to declare multiple variables in the same statement, they must share
                              // the same type declaration and not repeat it, thus this will work:

double d1, d2;                // is legal
```

1.7.2 Identifiers

- Three rules for valid identifiers:
  - The name must begin with a letter or the symbol $ or _.

    - Subsequent characters may also be numbers.
    - You cannot use the same name as a Java reserved keyword.
- List of Java reserved keywords (no need to memorize them for the exam, it will only ask about the ones you've already learned) – (* `const` and `goto` aren't actually used in Java):

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

- Consistency when declaring variable names is that it starts with a lowercase letter and then uses CamelCase, i.e. each word begins with an uppercase letter.
- Method and variable names begin with a lower case letter followed by CamelCase.
- Class names begin with an uppercase letter followed by CamelCase. Don't start any identifier with $ (although it is allowed), because the compiler uses this symbol for some files.

1.8 Understanding Default Initialization of Variables

1.8.1 Local Variables

- A local variable is a variable defined within a method.
- They must be initialized before use.
- They do not have a default value and contain garbage data until initialized.
- The compiler will not let you read an uninitialized value, e.g. `int y = 10; int x; int reply = x + y;` does not compile. Until `x` is assigned a value, it cannot appear within an expression.

1.8.2 Instance and Class Variables

- Variables that are not local variables are known as fields of the class, which can either be:
    - instance variables
    - class variables, which are shared across multiple objects and can be recognized by its keyword `static` in front of its name
- Instance and class variables do not require you to initialize them. As soon as they are declared they get a default value which is `null` for an object and `0`/`false` for a primitive:

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`

1.9 Understanding Variable Scope

- Method parameters (variables passed into a method) are local to the method.
- Method parameters and variables declared inside a method have local scope to the method.
- Local variables can never have a scope larger than the method they are defined in.
- They can even have a smaller scope if they are declared within a block inside a method; each block of code, i.e. code between curly brackets – `{}` –, has its own scope. When there are multiple blocks, you match them from

the inside out. These smaller contained blocks can reference variables defined in the larger scoped blocks, but not vice versa.

- Instance variables are available as soon as they are defined and last for the entire lifetime of the object itself (until object is garbage collected).
- Class (static) variables go into scope when declared and stay in scope for the entire life of the program.

1.10 Ordering Elements in a Class

- Comments can go anywhere in the code.
- Other 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

- Fields and methods must be within a class.
- Multiple classes can be defined in the same file, but only one of them is allowed to be `public`. The `public class` matches the name of the file.
- A file is also allowed to have neither class be `public`. As long as there isn't more than one `public class` in a file, it is okay.

1.11 Destroying Objects

Java provides a garbage collector to automatically look for objects, which are stored on the memory's heap, that aren't needed anymore.

1.11.1 Garbage Collection

- This is 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; it suggests that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.
- You should be able to recognize when objects become eligible for garbage collection.
- Java waits patiently until the code no longer needs that memory. An object will remain on the heap until it is no longer reachable, which is when:
  - The object no longer has any references pointing to it.
  - All references to the object have gone out of scope.
- Objects vs. References:
  - Do not confuse a reference with the object that it refers to; they are two different entities.
  - The reference is a variable that has a name and can be used to access the contents of an object.
  - A reference can be assigned to another reference, passed to a method, or returned from a method.
  - An object sits on the heap and does not have a name, thus you can only access it through a reference.
  - An object cannot be assigned to another object, nor can an object be passed to a method or returned from a method.
  - It is the object that gets garbage collected, not its reference.

1.11.2 finalize()

- The `finalize()` method gets called if the garbage collector tries to collect the object.
- For the exam you need to know `finalize()` could run zero or one time: it might not get called and it definitely won't be called twice: if the garbage collector doesn't run `finalize()` will not be called, if the garbage collector fails, it will not be called a second time.

1.12 Benefits of Java

- ***Object oriented:*** all code is defined in classes and most of those can be instantiated into objects. Java allows for functional programming within a class, but object oriented is still the main organization of code.
- ***Encapsulation:*** Java supports access modifiers to protect data from unintended access and modification.
- ***Platform Independent:*** write once, run everywhere.
- ***Robust:*** prevents memory leaks (like in C++) since it manages memory on its own (automatic garbage collection).
- ***Simple:*** eliminates pointers and got rid of operator overloading (like in C++ `a + b` could have any meaning).
- ***Secure:*** runs inside JVM, which creates a kind of sandbox making it hard for Java code to do evil things to the computer it is running on.

1.13 Exam Essentials

- ***Be able to write code using a main() method:*** `public static void main(String[] args)`. Args are 0-indexed, so first argument is accessed by `args[0]`. Accessing an argument not passed in causes an exception.
- ***Understand the effect of using packages and imports:*** packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not add subdirectories. In case of conflict, class name imports take precedence.
- ***Be able to recognize a constructor:*** has the same name as the class looking like a method but without a return type.
- ***Be able to identify legal and illegal declarations and initialization:*** multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization; others use the default value for that type. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers. Numeric literals may contain underscores between two digits and begin with 1-9, 0, 0x, 0X, 0b, and 0B.
- ***Be able to determine where variables go into and out of scope:*** all variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is garbage collected. Class (static) variables remain in scope as long as the program runs.
- ***Be able to recognize misplaced statements in a class:*** package and import statements are optional, but if present both go before the class declaration in that order, i.e. first package name, then import statements, then class declaration (which is mandatory). Fields and methods are also optional and are allowed in any order within the class declaration.
- ***Know how to identify when an object is eligible for garbage collection:*** draw a diagram to keep track of reference and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

2 Operators and Statements

2.1 OCA Exam Objectives

1. Using Operators and Decision Constructs:
    a. Use Java operators; including parentheses to override operator precedence.
    b. Create if and if/else and ternary constructs.
    c. Use a switch statement.
2. Using Loop Constructs:
    a. Create and use while loops.
    b. Create and use for loops including the enhanced for loop.
    c. Create and use do/while loops.
    d. Compare loop constructs.
    e. Use break and continue.

2.2 Understanding Java Operators

- A Java operator is a symbol that can be applied to a set of variables, values, or literals (operands) that returns a result.
- Java knows unary, binary and ternary operators which can be applied to one, two, or three operands respectively.
- Java operators are not necessarily evaluated from left-to-right; unless overridden with parentheses, they follow order of operation (operator precedence) as follows (for OCA exam the ones in bold are relevant):

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 : expression2***
Assignment operators	***=***, ***+=***, ***-=***, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

- Example: `int y = 4; double x = 3 + 2 * --y;`
  First y is decremented to 3, then multiplied by 2 resulting in 6, then added with 3 resulting in 9 upcasted to 9.0.

2.3 Working with Binary Arithmetic Operators

Binary operators (most common) perform mathematical operations on variables, create logical expressions and perform basic variable assignments. They are usually combined in complex expressions with more than two variables, thus operator precedence is very important in evaluating expressions.

2.3.1 Arithmetic Operators

- These include addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`) and modulus (`%`).
- They also include unary operators `++` and `––`.
- Multiplicative operators (`*`, `/`, `%`) have higher order of precedence than additive operators (`+`, `–`):
  `int x = 2 * 5 + 3 * 4 – 8;` → `int x = 10 + 12 – 8;` → remaining terms are evaluated in left-to-right order resulting in `14`.
- Order of operation can be changed by using parentheses around the sections you want to evaluate first:

`int x = 2 * ((5 + 3) * 4 – 8);` → `int x = 2 * (8 * 4 – 8);` → `int x = 2 * (32 – 8);` → `int x = 2 * 24` resulting in `48` for `x`.

- All of the arithmetic operators may be applied to any Java primitives, except boolean.
- Only `+` and `+=` may be applied to `String` values (`String` concatenation).
- The modulus (remainder) operator is the remainder when two numbers are divided, e.g. `9 / 3` divides evenly into `3`, thus `9 % 3 = 0`; but `11 / 3` doesn't divide evenly, thus `11 % 3 = 2`.
- For integer values, division results in the floor value of the nearest integer that fulfills the operation, whereas modulus is the remainder value.
- For a given divisor `y`, the modulus operation results in a value between `0` and `(y – 1)` for positive dividends.
- The modulus operation may also be applied to negative integers and floating-point numbers; for a given divisor `y` and negative dividend, the resulting modulus value is between (`-y + 1`) and `0` (this is not relevant for the exam).

2.3.2 Numeric Promotion

- 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 (`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` (not the case for unary operator).
- 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.
- Examples:

```
int x = 1; long y = 33; x * y;                // result data type is a long
double x = 39.21; float y = 2.1; x + y;       // does not compile: should be float y = 2.1f; then
                                              // result data type would be double
short x = 10; short y = 3; x / y;             // x and y will be promoted to int before the operation
short x = 14; float y = 13; double z = 30;    // all rules apply: x promotes to int, then to float so
  x * y / z;                                  // it can multiplied by y (in this case it doesn't have
                                              // to be 13f), x * y promotes to double so that it can be
                                              // divided with z, resulting in datatype double
```

2.4 Working with Unary Operators

A unary operator requires exactly one operand, or variable to function, e.g. for increasing a numeric variable by one, or negating a boolean variable.

Unary operator	Description
`+`	Indicates a number is positive, which is the default 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.

2.4.1 Logical Complement and Negation Operators

- The logical complement operator (`!`) flips the value of a boolean expression: true becomes false, false becomes true.
- The negation operator (–) reverses the sign of a numeric expression.
- Some operators require the variable or expression they're acting upon to be of a specific type: the negation operator cannot be applied to a boolean expression (false is not equal to 0 and true is not equal to 1) and a logical complement operator cannot be applied to a numeric expression. **Watch out for this on the exam!**

2.5 Increment and Decrement Operators

- Increment (++) and decrement (−−) operators can be applied to numeric operands and have the higher order or precedence, as compared to binary operators.
- They often get applied to an expression.
- Pre-increment and pre-decrement operator: the operator is placed before the operand; the operator is applied first and the value returned is the new value of the expression.
- Post-increment and post-decrement operator: the operator is placed after the operand; the original value of the expression is returned and the operator is applied after the value is returned.
- **Watch out on the exam:** multiple increment or decrement operators can be applied to a single variable on the same line, e.g.

```
int x = 3; int y = ++x * 5 / x-- + --x;
int y = 4 * 5 / x-- + --x;             // x assigned value of 4
int y = 4 * 5 / 4 + --x;               // x assigned value of 3
int y = 4 * 5 / 4 + 2;                 // x assigned value of 2

Result is: x is 2 and y is 7.
```

2.6 Using Additional Binary Operators

- An assignment operator is a binary operator that modifies, or assigns, the variable on the left-hand side of the operator, with the result of the value on the right-hand side of the equation, e.g.: `int x = 1;` assigns 1 to x.
- Java automatically promotes from smaller to larger data types; it throws a compiler exception if it detects you are trying to convert from larger to smaller data types.

2.6.1 Casting Primitive Values

- We can assign a larger numerical data type to a smaller numerical data type by applying casting or by converting from a floating-point number to an integral value, e.g.:

```
int x ((int)1.0;
short y = (short)1921222;       // too large to store (numeric overflow); stored as 20678
int z = (int)9f;
long t = 192301398193810323L;
```

- Overflow is when a number is so large that it will no longer fit within the data type; the number “wraps around” to the next lowest value and counts up from there.
- When a number is too low to fit in the data type, we speak of underflow (out of scope for the exam).
- The following does not compile:

```
short x = 10;
short y = 3;
short z = x * y;      // DOES NOT COMPILE because x and y are promoted to int causing z to be an int.
```

To make it compile, we need to cast the result of the multiplication to a short:

```
short z = (short)(x * y);       // tells the compiler to ignore its default behavior and you need to
                                // take care to prevent overflow or underflow
```

2.6.2 Compound Assignment Operators

- Only two for the OCA exam are required to know: += and −=.
- Examples:

```
int x = 2, z = 3;
x = x * z;            // simple assignment operator
x *= z;               // compound assignment operator: result is the same as with the simple assignment
                      // operator (it’s a shorthand notation)
```

- 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.
- The shorthand notation can save us from having to explicitly cast a value:

```
long x = 10; int y = 5;
y = y * x;              // DOES NOT COMPILE
y = (int)(y * x);       // DOES COMPILE
y *= x;                 // DOES COMPILE: compound operator will first cast y to long, apply multiplication
                        // of two long values, and then cast the result to an int
```

- The result of the assignment operator is an expression in and of itself, equal to the value of the assignment:

```
long x = 5; long y = (x = 3);
System.out.println(x);    // outputs 3
System.out.pringln(y);    // also outputs 3: sets value of variable x to 3, then returns value of
                          // assignment, which is also 3
```

2.6.3 Relational Operators

- Relational operators compare two expressions and return a boolean value.
- Relational operators are (for the examples: `int x = 10, y = 20, z = 10;`):

Relational operator	Description	Example	Result
`<`	Strictly less than.	`x < y`	true
`<=`	Less than or equal to.	`x <= y`	true
`>`	Strictly greater than.	`x >= z`	true
`>=`	Greater than or equal to.	`x > z`	false
`a instanceof b`	True if reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b. Out of scope for OCA exam.		

- The first four from above table are applied to numeric primitive data types; if the two numeric operands are not of the same data type, the smaller one is promoted as discussed before.
- The instanceof is applied to object references and classes or interfaces.

2.6.4 Logical Operators

- Logical operators, (`&`), (`|`) and (`^`), may be applied to both numeric and boolean data types.
- Called logical operators when applied to boolean data types.
- Called bitwise operators when applied to numeric data types. For the OCA exam you don't need to know anything about numeric bitwise comparison.
- Logical truth tables for `&`, `|`, and `^`:

x & y
(AND)

	y = true	y = false
x = true	true	false
x = false	false	false

x | y
(INCLUSIVE OR)

	y = true	y = false
x = true	true	true
x = false	true	false

x ^ y
(EXCLUSIVE OR)

	y = true	y = false
x = true	false	true
x = false	true	false

  - And is only true if both operands are true.
  - Inclusive OR is only false if both operands are false.
  - Exclusive OR is only true if the operands are different.
- Conditional operators, (`&&`) and (`||`), are also called short-circuit operators: right-hand side of the expression may never be evaluated if the final result can be determined by left-hand side of the expression, e.g.:

```
if (x != null && x.getValue() < 5) { … }       // if x is null, x.getValue() < 5 is never evaluated
if (x != null & x.getValue() < 5) { … }        // both will be evaluated, so if x is null, x.getValue()
                                               // will  throw NullPointerException
```

- **Be wary** of short-circuit behavior on the exam, as questions are known to alter a variable on the right-hand side of the expression that may never be reached.

2.6.4.1 Equality Operators

- There is a semantic difference between "two objects are the same" and "two objects are equivalent".
- For numeric and boolean primitives, there is no such distinction.

- The equals operator (==) and not equals operator (`!=`) compare two operands and return a boolean value whether the expressions or values are equal, or not equal, respectively.
- Equality operators can be used in three scenarios:
    - Comparing two numeric primitive types: if numeric values are of different data types, the values are automatically promoted as described earlier, e.g. `5 == 5.00` returns `true`.
    - Comparing two boolean values.
    - Comparing two objects, including `null` and `String` values.
- You cannot mix and match types, e.g.: `boolean x = true == 3;` does not compile.
- **Pay close attention** to the data types when you see an equality operator on the exam; assignment operators and equality operators are often mixed, e.g.:

```
boolean y = false; boolean x = (y = true);
System.out.prinln(x);                            // outputs true
```

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

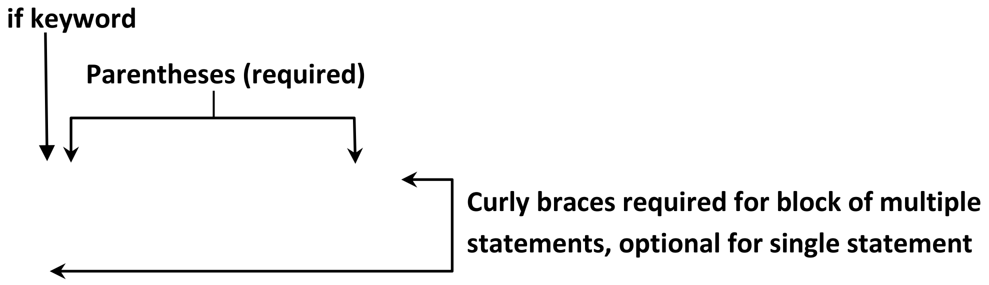
```
File x = new File("a.txt"); File y = new File("a.txt"); File z = x;
System.out.println(x == y);                      // outputs false
System.out.println(x == z);                      // outputs true
```

2.7 Understanding Java Statements

- A Java statement is a complete unit of execution in Java, terminated with a semicolon (`;`).
- 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.
- Statements can be applied to single expressions as well as a block of Java code, the latter being a group of zero or more statements between balanced braces (`{}`) which can be used anywhere a single statement is allowed.

2.7.1 The if-then Statement

- We use the `if-then` statement if we only want to execute a block of code under certain circumstances, i.e. it will be executed if and only if a boolean expression evaluates to `true` at runtime.
- The structure of an `if-then` statement:

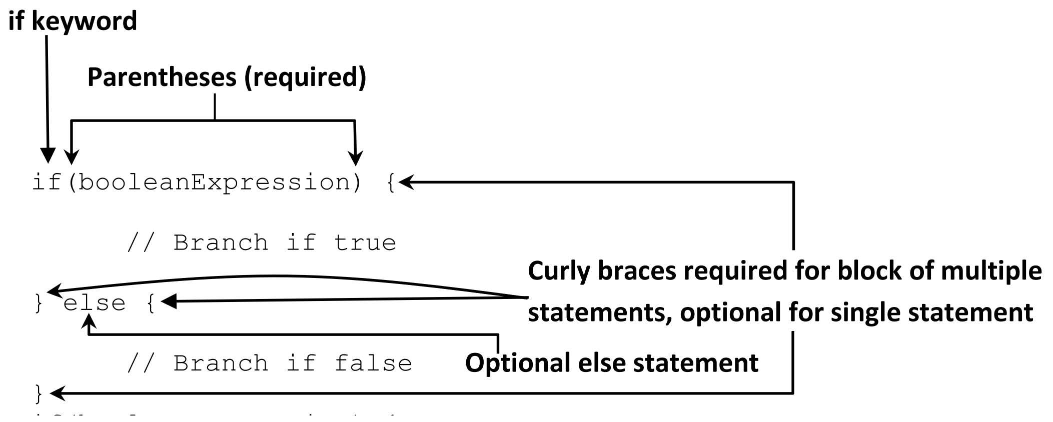

if keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

- A statement block allows multiple statements to be executed based on the `if-then` evaluation. For readability it is good practice to put blocks around the execution component of `if-then` statements (even if it is only one statement).
- **On the exam watch out for if-then statements without braces (`{}`).** Be sure to trace the open and close braces of the block and ignore any indentation you may come across.

2.7.2 The if-then-else Statement

- To branch between one of two possible options, we could write two different statements, but a better way is to use the `if-then-else` statement in which the boolean evaluation is done only once.
- The `else` operator takes a statement or block of statements.

- The structure of an `if-then-else` statement:



- We can append additional `if-then` statements to an `else` block. Java process will continue execution until it encounters an `if-then` statement that evaluates `true` and if none found it will execute the final code of the `else` block, e.g.:

```
if (…) {
   // statements
} else if (…) {
   // statements
} else {
   // statements
}
```

- Note though that in the above example of a nested `if-then-else`, order is important. You need to be careful not to create unreachable code, e.g.:

```
if (hourOfDay < 15) {
   // statements
} else if (hourOfDay < 11) {
   // statements      // UNREACHABLE CODE: only one branch can be executed here – if the first branch is
                       // executed, then the second cannot be executed
} else {
   // statements
}
```

- **On the exam watch out for code where the boolean expression inside the `if-then` statement is not actually a boolean expression:** `int x = 1; if (x) { … }` does not compile.
- **Also be wary of assignment operators being used as if they were `equals` (==) operators in `if-then` statements:** `int x = 1; if (x = 5) { … }` does not compile.

2.7.3 Ternary Operator

- The ternary operator, or conditional operator, `? :`, takes three operands in the form of:

  `booleanExpression ? expression₁ : expression₂`

- The first operand must be a boolean expression, the second and third can be any expression that returns a value.
- It is a condensed form of an `if-then-else` statement that returns a value.
- The second and third expressions don't have to be of the same data types, but it may come into play when combined with the assignment operator:

```
System.out.println((y > 5) ? 21 : "Zebra";   // COMPILES: the method System.out.println can convert
                                             // both expressions to String
int animal = (y < 91) ? 9 : "Horse";         // DOES NOT COMPILE: String Horse cannot be assigned to int
```

- Only one of the right-hand expressions of the ternary operator will be evaluated at runtime. Like short-circuit operators, if one of the two right-hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime, in other words: the expressions in a ternary operator may not be applied if the particular expression is not used. **At the exam, be wary of any question that includes a ternary expression in which a variable is modified in one of the right-hand side expressions:** e.g.:

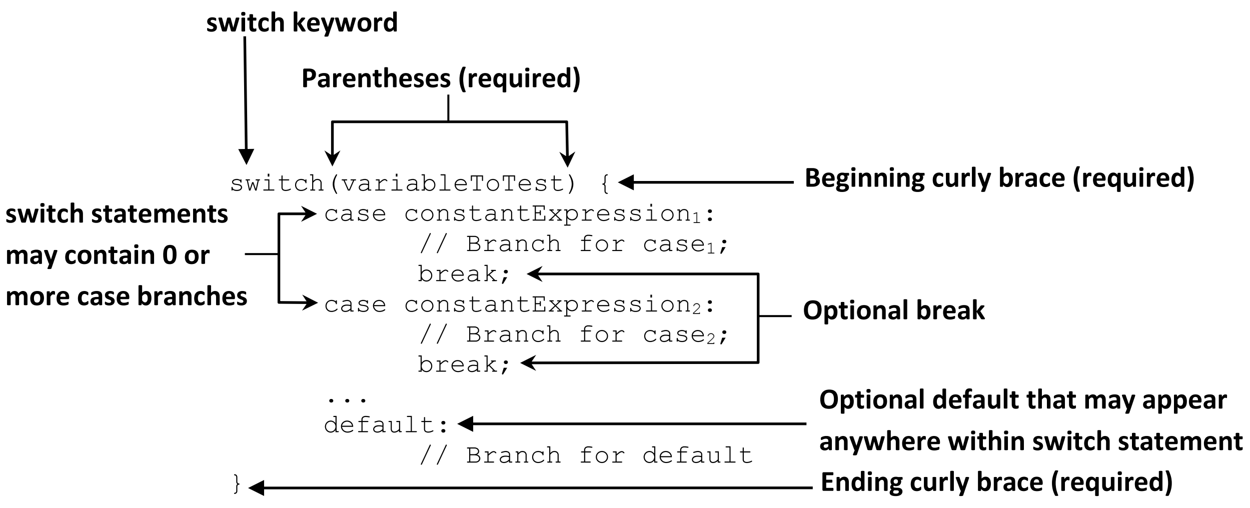
```
int y = 1; int z = 1;
final int x = y<10 ? y++ : z++;     // left-hand boolean expression is true, so only y is incremented
                                    // resulting in y having value of 2 and z value of 1
int y = 1; int z = 1;
final int x = y>=10 ? y++ : z++;    // left-hand boolean evaluates to false, so only z is incremented
                                    // resulting in y having value of 1 and z value of 2
```

2.7.4 The switch Statement

- The switch-statement is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch (case statement).
- If a case statement is found that matches the value, an optional `default` statement will be called.
- If no such `default` option is available, the entire `switch` statement will be skipped.

2.7.4.1 Supported Data Types

- A `switch` statement has a target variable that is not evaluated until runtime.
- Data types supported by switch statements are:
    - byte and Byte
    - short and Short
    - char and Character
    - int and Integer
    - String
    - enum values
- The structure of a `switch` statement:



```
switch(variableToTest) {
    case constantExpression1:
        // Branch for case1;
        break;
    case constantExpression2:
        // Branch for case2;
        break;
    ...
    default:
        // Branch for default
}
```

switch keyword
Parentheses (required)
Beginning curly brace (required)
switch statements may contain 0 or more case branches
Optional break
Optional default that may appear anywhere within switch statement
Ending curly brace (required)

2.7.4.2 Compile-time Constant Values

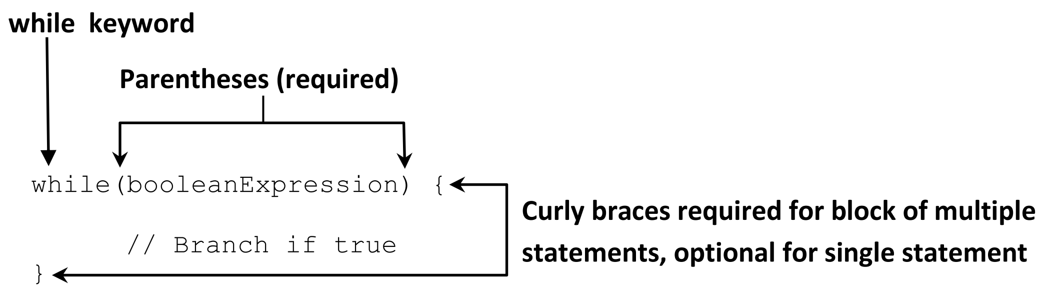
- The values in each `case` statement must be compile-time constant values of the same data type as the `switch` value, thus only literals, `enum` constants, or `final` constant variables of the same data type can be used.
- A `final` constant is marked with the `final` modifier and initialized with a literal value in the same expression in which it is declared.
- A `break` statement at the end of a `case` or `default` statement will terminate the `switch` statement and return flow control to the enclosing statement.
- If you leave out the `break` statement, flow will continue to the next proceeding `case` or `default` block automatically.
- The `case` or `default` statements don't have to be in any particular order, unless you are going to have pathways that reach multiple sections of the `switch` block in a single execution.
- The `default` block is only branched to if there is no matching `case` value for the `switch` statement, regardless of its position within the `switch` statement.

- While the code will not branch to the `default` statement if there is a matching `case` value within the `switch` statement, it will execute the `default` statement if it encounters it after a `case` statement for which there is no terminating `break` statement (**in this case order of the `default` and `case` statements does matter!**).
- **The exam creators are fond of `switch` examples that are missing `break` statements!** Always consider that multiple branches may be visited in a single execution.
- The data type for `case` statements must all match the data type of the `switch` variable.
- Examples:

```
private int getSortOrder(String firstName, final String lastName) {
   String middleName = "Patricia";
   final String suffix = "JR";
   int id = 0;
   switch(firstName) {
      case "Test":        // COMPILES: using String literal
         return 52;       // return statement also exits the switch (like a break)
      case middleName:    // DOES NOT COMPILE: middleName is not final
         id = 5; break;
      case suffix:        // COMPILES: suffix is a final constant variable
         id = 0; break;
      case lastName:      // DOES NOT COMPILE: is final, but not a constant (passed into function)
         id = 8; break;
      case 5:             // DOES NOT COMPILE: String does not match with data type of switch variable
         id = 7; break;
      case 'J':           // DOES NOT COMPILE: char does not match with data type of switch variable
         id = 10; break;
      case java.time.DayOfWeek.SUNDAY:  // DOES NOT COMPILE: enum does not match with data type of
         id = 15; break;                // switch variable
   }
   return id;
}
```

2.7.5 The while Statement

- A repetition control structure (loop) executes a statement of code multiple times in succession.
- By using nonconstant variables, each repetition of the statement may be different.
- The `while` statement is the simplest repetition control structure.
- It has a termination condition, implemented as a boolean expression, which continues as long as the expression evaluates to `true`.
- The structure of a `while` statement:

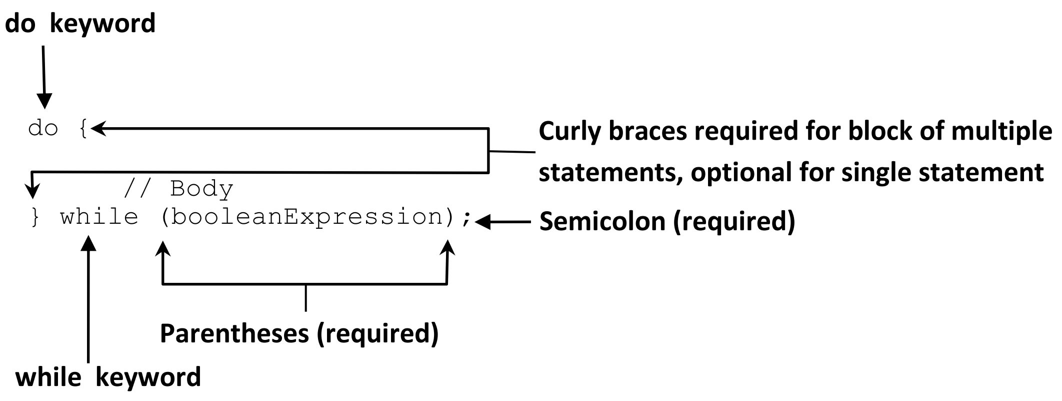

while keyword
Parentheses (required)
Curly braces required for block of multiple statements, optional for single statement

```
while(booleanExpression) {
      // Branch if true
}
```

- During execution, the boolean expression is evaluated before each iteration of the loop and exits if the evaluation returns `false`.
- A `while` loop may terminate after its first evaluation of the boolean expression, meaning the statement block may never be executed.
- The termination condition of a `while` statement can consist of a compound boolean statement, e.g.:
  `while (x > 0 && y > 0) { … }.`
- Be careful not to create an infinite loop, like: `int x = 2, int y = 5; while(x<10) y++;`
  You should be absolutely certain that the loop will eventually terminate under some condition: make sure the loop variable is modified, then ensure that the termination condition will be eventually reached in all circumstances.

2.7.6 The do-while Statement

- The `do-while` loop, like the `while` loop, also has a termination condition and statement, or block of statements, but it is guaranteed to be executed at least once, because it first executes the statement(s) and then checks the loop condition.
- The structure of a `do-while` statement:


do keyword
Curly braces required for block of multiple statements, optional for single statement
Semicolon (required)
Parentheses (required)
while keyword

```
do {
        // Body
} while (booleanExpression);
```

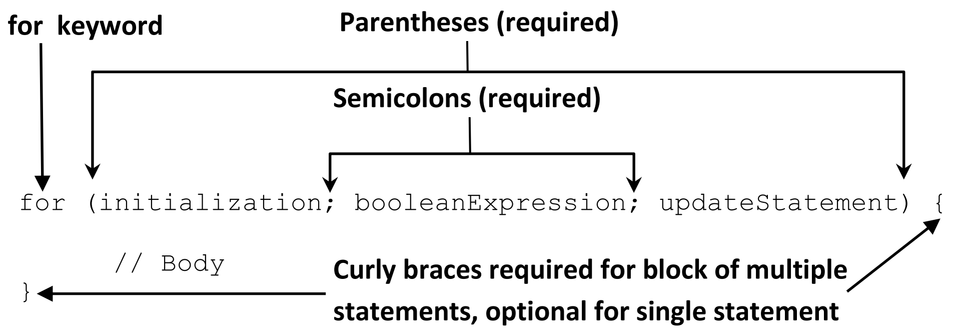
- Java recommends you use a `while` loop when a loop might not be executed at all and a `do-while` loop when the loop is executed at least once.

2.7.7 The for Statement

There are two variations of the `for` statement: the basic `for` loop and the enhanced `for` loop (`for-each` statement).

2.7.7.1 *The Basic for Statement*

- A basic `for` loop has the same conditional boolean expression and statement, or block of statements, as the other loops. Additionally, it also has an initialization block and an update statement.
- The structure of a basic `for` statement:


for keyword
Parentheses (required)
Semicolons (required)
Curly braces required for block of multiple statements, optional for single statement

```
for (initialization; booleanExpression; updateStatement) {
        // Body
}
```

1. Initialization statement executes.
2. If booleanExpression is `true` continue, else exit loop.
3. Body executes.
4. Execute updateStatements.
5. Return to Step 2.

- Each section is separated by semicolon. 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. **Watch out for this on the exam!**
- 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 `for` loop.
- The boolean condition is evaluated on every iteration of the loop before the loop executes.
- The components of the `for` loop are each optional. Leaving them all out will create an infinite loop. The semicolons separating the three sections are required though, e.g.:

```
for ( ; ; ) { System.out.println("Hello World"); }     // valid for loop, but infinite loop
for ( ) { … }                                          // will not compile
```

- Multiple terms can be added to the `for` statement:

```
int x = 0;                      // variable x can be declared before the loop begins
for (long y = 0, z = 4;         // init/boolean expression/update statements can include extra variables
     x < 5 && y < 10;           //   that may not reference each other: z is initialized but never used
     x++, y++)  {               // update statement can modify multiple variables
   System.out.print(y + " ");   // prints: 0 1 2 3 4
}
System.out.println(x);          // x can be used after the loop ended (declared outside loop) (prints: 5)
```

- It's not possible to redeclare a variable in the initialization block (this results in a compilation error):

```
int x = 0;
for (long y = 0, x = 4;          // DOES NOT COMPILE: x is already declared before the loop and repeated
     x < 5 && y < 10;            // in the initialization block > duplicate variable declaration
     x++, y++) {
}

int x = 0; long y = 10;
for (y = 0, x = 4;               // DOES COMPILE: init assigns value to x but does not declare it
     x < 5 && y <10;
     x++, y++) {
}
```

- It's not possible to use incompatible data types in the initialization block:

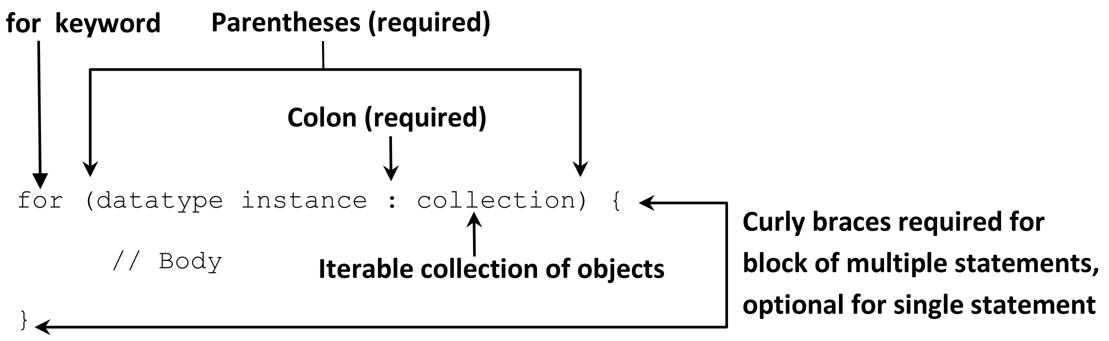
```
for (long y = 0, int x = 4;      // DOES NOT COMPILE: variables in initialization block must all be of
     x < 5 && y < 10;            // the same type, so either both y and x should be long or int
     x++, y++) {
}
```

- It's not possible to use loop variables outside the loop:

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

2.7.7.2 The for-each Statement

- The `for-each` loop is specifically designed for iterating over arrays and `Collection` objects.
- The structure of an enhanced `for-each` statement:


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

- The `for-each` declaration consists of an initialization section and an object to loop 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` (most of the Java `Collections` framework).
- The left-hand side 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; the datatypes need to be the same, e.g.:

```
String[] names = new String[3];
for (int name : names) { … }    // DOES NOT COMPILE: names is an array of String there for the named
                                // variable needs to be a String also and not an int
```

- **For the OCA exam, the only members of the `Collections` framework you need to know are `List` and `ArrayList`.**
- 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, e.g.:

```
String names = "Lisa";
for (String name : names) { … }  // DOES NOT COMPILE: names is not an array / doesn't implement Iterable
```

- The `for-each` loop is convenient for working with lists, but it hides access to the loop iterator variable. For example if we wanted to print something only on the first occurrence of the loop, we would usually use a basic `for` loop where we have access to the iterator variable.
- 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, e.g. `values[i] - values[i-1]`.

2.8 Understanding Advanced Flow Control

2.8.1 Nested Loops

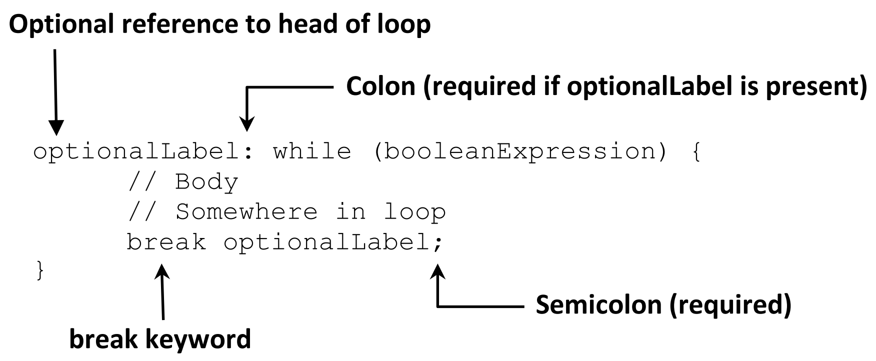
- Loops can contain other loops. For example, we can iterate over a two-dimensional array, an array that contains another array as its members.
- Nested loops can include `while` and `do-while`.

2.8.2 Adding Optional Labels

- `If-then` statements, `switch` statements and `loops` 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 word proceeded by a colon (`:`) and it's often used in loop structures.
- When dealing with only one loop, a label doesn't add any value. But they are useful when using nested loops.
- It is possible to add optional labels to control and block structures (this is not on the OCA exam and it's not good coding practice).
- Labels are usually in uppercase, with underscores between words (to distinguish them from regular variables).

2.8.3 The break Statement

- A `break` statement transfers the flow of control out to the enclosing statement. The same goes for `break` statements appearing inside of `while`, `do-while`, and `for` loops, as it will end the loop early.
- The structure of a `break` statement:



```
optionalLabel: while (booleanExpression) {
       // Body
       // Somewhere in loop
       break optionalLabel;
}
```

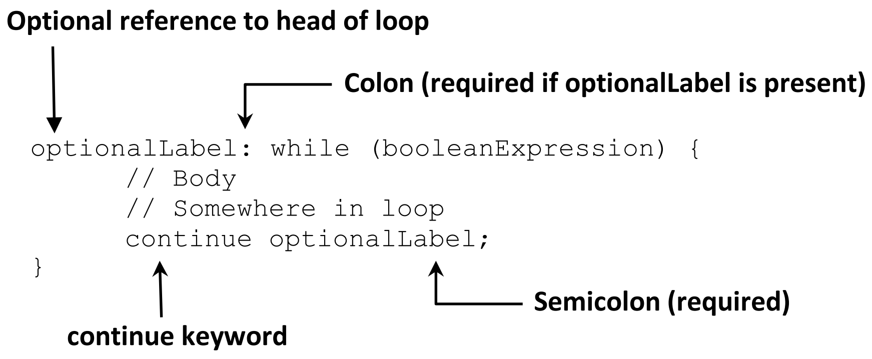
Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
break keyword

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

2.8.4 The continue Statement

- The `continue` statement causes the flow to finish the execution of the current loop.

- The structure of a `continue` statement:


Optional reference to head of loop
Colon (required if optionalLabel is present)
Semicolon (required)
continue keyword

```
optionalLabel: while (booleanExpression) {
      // Body
      // Somewhere in loop
      continue optionalLabel;
}
```

- 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.
- The `continue` statement, like the `break` statement, is applied to the nearest inner loop under execution using optional label statements to override this behavior.
- Advanced flow control usage table:

	Allows optional labels	**Allows unlabeled break**	**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 `if-then` statements.

2.9 Exam Essentials

- ***Be able to write code that uses Java operators.***
- ***Be able to recognize which operators are associated with which data types:*** some are only applied to numeric primitives, some only to boolean values, and some only to objects. It's important to notice when operator and operand(s) mismatch.
- ***Understand Java operator precedence.***
- ***Be able to write code that uses parentheses to override operator precedence.***
- ***Understand if and switch decision control statements:*** these often appear in exam questions unrelated to decision control.
- ***Understand loop statements.***
- ***Understand how break and continue can change flow control.***

Appendix B – Study Tips

Taking the test

Reviewing Common Compiler Issues

Common tips to determine if Code Compiles:

- Keep an eye out for all reserved words. [Chapter 1]
- Verify brackets – `{}` – and parentheses – `()` – are being used correctly. [Chapter 1]
- Verify `new` is used appropriately for creating objects. [Chapter 1]
- Ignore all line indentation especially with `if-then` statemnets that do not use brackets `{}`. [Chapter 2]
- Make sure operators use compatible data types, such as the logical complement operator (`!`) only applied to boolean values, and arithmetic operators (`+`, `-`, `++`, `--`) only applied to numeric values. [Chapter 2]
- For any numeric operators, check for automatic numeric promotion and order or operation when evaluating an expression. [Chapter 2]
- Verify `switch` statements use acceptable data types. [Chapter 2]
- Remember `==` is not the same as `equals()`. [Chapter 3]
- String values are immutable. [Chapter 3]
- Non-void methods must return a value that matches or is a subclass of the return type of the method. [Chapter 4]
- If two classes are involved, make sure access modifiers allow proper access of variables and methods. [Chapter 4]
- Nonstatic methods and variables require an object instance to access. [Chapter 4]
- If a class is missing a default no-argument constructor or the provided constructors do not explicitly call `super()`, assume the compiler will automatically insert them. [Chapter 5]
- Make sure abstract methods do not define an implementation, and likewise concrete methods always define an implementation. [Chapter 5]
- You implement an interface and extend a class. [Chapter 5]
- A class can be cast to a reference of any superclass it inherits from or interface it implements. [Chapter 5]
- Checked exceptions must be caught; unchecked exceptions may be caught but do not need to be. [Chapter 6]
- `try` blocks require a `catch` and/or `finally` block for the OCA exam. [Chapter 6]