Chapter 1: Java Building Blocks.

Object: a runtime instance of a class in memory. All the various objects of all the different classes represent the state of your program.

Elements of a java class: methods and fields. They are the members of a class. Variables hold the state of the program and methods operate on that state.

Keyword: reserved words of the Java language.

Method signature: the full declaration of a method. E.g.: *public int numberVisitors (int month)*

There are two types of comments:

*// Single-line comment*

*/\**

*Multiple-line comment*

*\*/*

*/\* And*

*\* // they can be*

*\* combined*

*\*/*

Classes and files.

Classes are .java extension. Public classes are not required. You can put multiple classes in one file, at most one of those classes is allowed to be public. The name of the file needs to match the public class.

The main() method is the entry point of the Java program. This is managed by the JVM. The JVM calls the OS to allocate memory and CPU time, access files and so on.

*public class Zoo {*

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

*}*

**Public** is the access modifier (it can be protected or private)

**Static** binds the method to its class.

**Void** is the return type (can be any other type, primitive or user defined). It’s a good practice to use the a void method to change an object’s state.

**Main** is the name of the method.

Inside the parenthesis is the argument. **Args** is the name. **String** is an array of java.lang.String type. These are acceptable as well for main: String args[] or String… args.

To compile a java file: *javac Zoo.java.* The result will be a file of bytecode by the same name, but with a .class extension.

To run the file: *java Zoo*

To run passing parameters: java Zoo Bronx “San Diego” Zoo

Package declarations and imports.

Java come with thousands of built-in classes which are organized in packages. In order to use them, you would need to import them otherwise an error will be thrown.

There are a few things to keep in mind. Wildcards access all classes inside of a package. Java only looks at classnames in a package, it will not read / import classes contained in other sub-packages. Static imports can import other types.

A classname has priority over wildcard when importing multiple packages and classes.

Java.lang is automatically imported.

Java.nio.file for files and paths.

When the need to import different classes with the same naming convention, it’s a good approach to import one of them and use the other’s fully qualified name, or the fully qualified name for both.

*import java.util.Date;*

*public class Conflicts {*

*Date date;*

*java.sql.Date sqlDate;*

*}*

*public class Conflicts {*

*java.util.Date date;*

*java.sql.Date sqlDate;*

*}*

Importing both classes using the classnames or the wildcard, would be recognized by java as code error and will not compile.

Creating Objects.

To create an instance of a class, the keyword ‘new’ is used. E.g.: Random r = new Random();

Random() is a constructor. The purpose of the constructor is to initialize fields. It must match the name of the class and there isn’t any return type in the method signature.

Order of initialization: fields and instance initializers blocks are run in the order in which they appear in the file. However, the constructor runs after all fields and instance initializers blocks have run.

Instance Initializer example: *{ System.out.println(“setting constructor”); }*

Data types.

Java contains two types of data: primitives and reference types. There are 8 primitives types built in the Java language:

|  |  |  |  |
| --- | --- | --- | --- |
| boolean | true or false |  | true |
| byte | 8-bit integral | 28 = 2\*2=4\*28\*2=  16\*2=32\*2=64\*2= 128\*5=256  This means the range for byte is -128 to 127 | 123 |
| short | 16-bit integral | 216 | 123 |
| int | 32-bit integral | 232 | 123 |
| long | 64-bit integral | 264 | 3123456789L |
| float | 32-bit floating-point | 232 | 123.45f |
| double | 64-bit floating-point | 264 | 123.456 |
| char | 16-bit Unicode |  | ‘a’ |

You can now the max value of a primitive type by calling the MAX\_VALUE field. E.g.: System.out.println(Integer.MAX\_VALUE); and it will print *2,147,438,647*

A number defined in a variable is called a *literal*. E.g.: *long max = 3123456789L;*

In literals, you can use underscore for big numbers. This is NOT accepted by Java 🡪 *double a = \_1000\_.\_00\_;* They can go anywhere between the numbers in order to make easier the value to read. E.g.: *double a = 1\_000\_000.0\_0;*

*Base 10 numbers – decimal number system (0-9).*

*Base 8 numbers / octal (0-7), uses 0 as the prefix. E.g.: 017*

*Base 16 numbers / hexadecimal (0-9, A-F), uses I followed by x or X as a prefix. E.g.: 0xFF*

*Base 2 numbers / binary (0-1), uses 0 followed by b or B as a prefix. E.g.: 0b10*

Reference Types.

These types refer to an object (instance of a class). They hold the address where the object is located in memory – pointer. A value can be assigned in one of two ways: a reference can be assigned to another object of the same type or a new object using the new keyword.

Reference types can be assigned null value, while primitives cannot. Reference types can be used to call methods when they do not point to null. Primitives do not have methods.

Identifiers. There are three rules for identifiers’ names:

* The name must begin with a letter or the symbol $ or \_.
* Subsequent characters may also be numbers.
* You cannot use a java reserved word for this.

Understanding default initialization of variables.

Local variables are defined within a method. They do not have a default value.

Instance (also called fields) and class variables are not local. Class variables have the *static* keyword. These are not required to initialize them, they get assigned a default value in that case.

*boolean 🡪 false*

*byte, short, int, long 🡪 0*

*float, double 🡪 0.0*

*char 🡪 ‘\u0000’ (NUL)*

*all objects reference type 🡪 null*

Variable scope.

Always verify the scope of local, class and instance variables.

* Local variables – in scope from the declaration to the end of the block.
* Instance variables – in scope from the declaration until the object is garbage collected.
* Class variables – in scope from the declaration until the program ends.

Ordering elements in a class.

This the following order for the elements in a class:

*Package // may not be required*

*Import // may not be required – goes immediately after package*

*Class // required – goes immediately after import*

*{ fields and methods } // may not be required – anywhere inside the class*

*// comments – anywhere in the file*

Garbage collection.

All java objects are stored in the program memory’s heap. The heap (free store) is a large pool of unused memory allocated to your java app. The Garbage Collector deletes the objects from memory that are no longer reachable by the app.

System.gc(); 🡪 a request for Garbage Collector to run, but this request can be ignored by Java.

An object is not reachable when:

1. No reference points to it.
2. The reference to the object goes out of scope.

*Finalize()* can be implemented but is only run when the object is eligible for the garbage collector. The method can run zero or one time. If garbage collector fails to collect the object and runs a second time, finilaize() won’t be called again.

Benefits of java.

* Object oriented. Organized in classes.
* Encapsulation. Supports access modifiers to protect data from unintended access and modifications.
* Platform independent. Java code compiles to bytecode. Can be read by the JVM on any OS.
* Robust. Prevents memory leaks.
* Simple. No pointers, no operator overloading.
* Secure. Java code runs in the JVM, which creates a sandbox to execute the code.

Chapter 2: Operators and Statements.

Operator: a special symbol that can be applied to a set of variables, values or literals – referred as operands – and that returns a result. Operators can be: unary, binary and ternary. Unless overwritten by parenthesis, Java follows order of operation, then Java guarantees left-to-right evaluation. Order of operator precedence:

|  |  |
| --- | --- |
| Post-unary operators | *expression++, expression--* |
| Pre-unary operators | *++expression, --expression* |
| Other unary operators | *~, +, -, !* |
| Multiplication / Division / Modulus | *\*, /, %* |
| Addition / Substraction | *+, -* |
| Shift operators | *<<, >>, >>>* |
| Relational operators | *==, !=* |
| Logical operators | *&, ^, |* |
| Short-circuit logical operators | *&&, ||* |
| Ternary operators | *boolean expression ? expression1 : expression2* |
| Assignment operators | *=, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=* |

Arithmetic Operators.

Addition (+), subtraction (-), multiplication (\*), division (/) and modulus (%). They include ++ and -– unary operators. \*, /, % have a higher precedence than + and -. They can be applied to all primitives type except for boolean and String. When +, += is applied to String, it’s concatenation.

Numeric Promotion.

Rules:

* If two values have different data types, Java will automatically promote one of the values to the larger of the two.

*int x = 1; long y = 33; x \* y data type will be long.*

* 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.

*double x = 39.21; float y = 2.1f; x + y data type will be double.*

* Smaller data types (byte, short, char) are first promoted to int any time they’re used with a Java binary arithmetic operator, even if neither of the operand is int.

*short x = 10; short y =3; x / y data type will be int.*

* After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

*short x = 14; float y = 13; double z = 30;*

*x \* y / z data type result will be double. X is first promoted to int and then to float to be multiplied with y. The result will be promoted to double to be divided by z.*

Unary Operators.

Only requires one operand / variable to function.

|  |  |
| --- | --- |
| *+* | *Positive number* |
| *-* | *Negative number* |
| *++* | *Increments value by 1* |
| *- -* | *Decrements value by 1* |
| *!* | *Inverts a Boolean’s logic value* |

Logical Complement: !, flips the logical value of a Boolean.

Cannot be applied to a numeric variables.

Negation Operator: -, reverses the sign in a numeric expression.

Increment and Decrement Operators: ++, - -, to be applied to numeric operands and have higher precedence or order, as compared to binary operators. These can be:

* Pre-decrement: --variable
* Pre-increment: ++variable
* Post-decrement: variable- -
* Post-increment: variable++

In the scenario in which variable is 5, all the operations above produce a different output. Pre operators will increment or decrement the value first, then return the result. Post operators will return the result first, then increment or decrement the value.

Binary Operators.

* Assignment operators. Modifies the value of the variable with the value on the right-hand side.

E.g.: *int x = 1;*  
Java will automatically promote from smaller to larger data type. Error will be thrown if it’s attempted to assign larger to smaller data type.

E.g.:

*int x = 1.0;*

*short y = 1921222;*

*int z = 9f;*

*long t = 192301398193810323;*

Casting primate values. Examples above can be fixed by casting the large data types to smaller ones.

E.g.:

*int x = (int)1.0;*

*short y = (short)1921222; // value overflows to 20,678*

*int z = (int)9f;*

*long t = 192301398193810323L;*

*Example of underflow: System.out.println(2147483647+1); // prints -2147483648*

*Another casting example to override JVM behavior:*

*short x = 10; short y = 3;*

*short z = (short)(x \* y);*

* Compound Assignment Operators. Includes +=, -=, /=, \*=.

E.g.:

*int x = 2, z = 3;*

*x = x \* z; // simple assignment.*

*x \*= z; // Compound assignment.*

Compound operators can apply a cast of larger values to small automatically.

E.g.:

*long x = 10; int y = 5;*

*y \*= x;*

Another valid and not common compound assignment:

*long x = 5;*

*long y = (x=3);*

*System.out.println(x);*

*System.out.println(y);*

Both output 3.

* Rational Operators. Compares two expressions and returns a Boolean value. This applies only to numeric primitive data types. As other operators, the smaller operand will be promoted the larger data type.

E.g:

*int x = 10, y = 20, z = 10;*

*System.out.println(x < y); // true*

*System.out.println(x <= y); // true*

*System.out.println(x >= z); // true*

*System.out.println(x > z); // false*

* Logical Operators. These &, | and ^ are applied to numeric and Boolean primitive data types. When applied to Boolean is referred as logical operators. When applied to numeric values is referred as bitwise operators, because they perform bitwise comparisons of the bits that compose the number.

|  |  |  |
| --- | --- | --- |
| x & y (AND) | x | y (Inclusive OR) | X ^ y (Exclusive OR) |
| |  |  |  | | --- | --- | --- | |  | y = true | y = false | | |  |  |  | | --- | --- | --- | |  | y = true | y = false | | |  |  |  | | --- | --- | --- | |  | y = true | y = false | |
| |  |  |  | | --- | --- | --- | | x = true | True | False | | |  |  |  | | --- | --- | --- | | x = true | True | True | | |  |  |  | | --- | --- | --- | | x = true | False | True | |
| |  |  |  | | --- | --- | --- | | x = false | False | False | | |  |  |  | | --- | --- | --- | | x = false | True | False | | |  |  |  | | --- | --- | --- | | x = false | True | False | |

Short-circuit operators (&&) and (||) are very similar to the previous ones, expect that the right hand of the expression may never be evaluated.

E.g.: *boolean x = true || (y < 4);*

Other examples:

*If(x != null %% x.getValue() < 5) {*

*// Do something*

*}*

* Equality Operators.

Equals ( == ) and not Equals ( != ) operators are used in the following scenarios:

1. Comparing two numeric primitives types. If data types are different, the smaller will be promoted to the larger type.
2. Comparing two boolean values.
3. Comparing two objects, including null and String values.

**Java Statements.**

A statement in Java is a complete unit of execution, terminated with a semicolon ( ; ). Control flow statements break up the flow of executing by using decision making, looping and branching, allowing the application to selectively execute specific segments of code.

The if-then Statement.

We only want to execute a block of code under certain circumstances. This statement allows the code to execute if the boolean expression evaluates to true at runtime.

*if ( booleanExpression ) {*

*// Branch if true*

*}*

The if-then-else Statement.

We want to execute different blocks of code when our condition may have different outputs.

*if ( booleanExpression ) {*

*// Branch if true*

*} else {*

*// Branch if false*

*}*

Ternary Operator. It’s the only operator in java that takes 3 operands and is of the form:

*booleanExpression ? expression1 : expression2*

The first one must be a boolean expression, and the second and third can be any expression that returns a value.

The Switch Statement.

It’s a complex decision-making structure in which a single value is evaluated, and flow is redirected to the first matching branch, known as a ***case*** statement. If no ***case*** is found that matches the value, an optional ***default*** statement will be called. If no ***default*** option is available, the whole block will be skipped.

Supported data types: byte and Byte (wrapper class), short and Short (w. class), char and Character (w. class), int and Integer (w. class), String and enum values. Does not support boolean and long.

*switch(variableToTest) {*

*case constantExpression1:*

*// Branch for case1;*

*break;*

*case constantExpression2:*

*// Branch for case2;*

*break;*

*default:*

*// branch for default;*

*}*

Compile-time Constant values.

The values in each case statement must be compile-time contact values of the same data type as the switch value. This means only literals, enum constants or final constant variables of the same data type can be used.

The While Statement.

A repetition control structure / loop. It’s the simplest loop in Java.

*while(booleanExpression) { // Body }*

The body will execute as long as the boolean expression is true.

The do-while Statement.

Do-while loop is guaranteed that the body will execute at least once, since the boolean expression is evaluated at the end of the block:

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

The Basic for Statement.

This contains an initialization block, a boolean expression block as the previous loops, and an update statement block.

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

Variables declared in the initialization block has a scope limited to the loop.

The boolean expression is evaluated in ever iteration before the loop executes.

5 variations of this for statement to familiarize with:

1. Creating an infinite loop.

*for ( ; ; ) { System.out.println(“Hello World”);*

1. 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 + " "); }*

1. Redeclaring a variable in the initialization block.

*int x = 0;*

*for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) { // Does not compile*

*System.out.print(x + " "); }*

1. Using incompatible data types in the initialization block.

*for (long y = 0, int x = 4; x < 5 && y < 10; x++, y++) { // Does not compile*

*System.out.print(x + " "); }*

1. 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*

The For-each Statement / Enhanced loop.

It was specifically design for iterating over arrays and Collections objects. It’s composed of an initialization section and an object to be iterated over.

*for(datatype instance : collection) { // Body }*

The right-hand side of the loop must be a built-in java array or an object whose class implements java.lang.Iterable, which includes most of the Java Collections framework. The left-hand side must include a declaration of an instance of a variable, whose type matched the type of a member of the array or collection in the right-hand side. For each iteration, the instance will have a different value from the collection in the right-hand side.

Nested Loops.

Loops can be nested, one inside of the other. And optional labels can be applied in order to make the code more understandable.

*OUTER\_LOOP: for(datatype instance : collection) {*

*INNER\_LOOP: for(initialization; booleanExpression; updateStatement) {*

*// Body*

*}}*

*while(booleanExpression) {*

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

*// Body }*

The Break Statement.

It transfers the flow of control out to the enclosing statement. It applies to all kinds of loop seen previously.

*optionalLabel : while (booleanExpression) {*

*// Body*

*break optionalLabel;*

*}*

Without the label, break will terminate the nearest inner loop. The optional label allows us to break out of a higher-level outer loop.

The Continue Statement.

It causes the flow to finish the execution of the current loop.

*optionalLabel : while (booleanExpression) {*

*// Body*

*continue optionalLabel;*

*}*

The continue statement transfers the control to the boolean expression that determines if the loop should continue. It only ends the current iteration of the loop. As the break statement, it is applied to the nearest inner loop, unless optional label is used.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Allows optional label | 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 |

Chapter 3: Core Java APIs.