TIP: If you are curious as to what Unicode characters are “letters” as far as Java is concerned, you can use the isJavaIdentifierStart and isJavaIdentifierPart methods in th**e Character** class to check.

TIP: Even though $ is a valid Java letter, you should not use it in your own code. It is intended for names that are generated by the Java compiler and other tools. Any currency symbol could be used as identifier.

Note that integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

double y = Math.sqrt(x);

System.out.println(y); // prints 2.0

NOTE: There is a subtle difference between the println method and the sqrt method.The println method operates on the System.out object. But the sqrt method in the Math class does not operate on any object. Such a method is called a static method.

When a number is present in the code, it is called a literal.

A local variable is a variable defined within a method. Local variables must be initialized before use.

Think of the acronym PIC (picture): package, import, and class. Fields and methods are easier to remember because they merely must be inside of a class.

**Objects vs. References**

Do not confuse a reference with the object that it refers to; they are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. **All references are the same size**, no matter what their type is. An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another

**finalize()**

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

Simple Java was intended to be simpler than C++. In addition to eliminating pointers, it got rid of operator overloading. In C++, you could write a + b and have it mean almost anything. But Java still has an overloaded operator +, which could be used either for numbers or for concatenation.

Understand the effect of using packages and imports. Packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not look at subdirectories. In the event of a import conflict, class name imports take precedence over wildcard \*.

**Overflow vs Underflow**

Overflow is when the absolute value of the number is too high for the computer to represent it.

Underflow is when the absolute value of the number is too close to zero for the computer to represent it.

You can get overflow with both integers and floating-point numbers. You can only get underflow with floating point numbers.

To get an overflow, repeatedly multiply a number by ten. To get an underflow repeatedly divide it by ten.

If the variable x is a signed byte it can have values in the range -128 to +127, then

x = 127

x = x + 1

will result in an overflow. +128 is not a valid value for x.

For floating point numbers, the range depends on their representation. If x is a single precision (32-bit IEEE) number, then

x = 1e-38

x = x / 1000

will result in an underflow. 1e-42 is not a valid value for x.

int x = 2, z = 3;

x = x \* z; // Simple assignment operator

x \*= z; // Compound assignment operator

The left-hand side of the compound operator can only be applied to a variable that is already defined and cannot be used to declare a new variable. In the previous example, if x was not already defined, then the expression x \*= z would not compile.

Exclusive OR (^) is only true if the operands are different.

**Comparing for and for-each Loops**

Since for and for-each both use the same keyword, you might be wondering how they are related. While this discussion is out of scope for the exam, let’s take a moment to explore how for-each loops are converted to for loops by the compiler. When for-each was introduced in Java 5, it was added as a compile-time enhancement. This means that Java converts the for-each loop into a standard for loop during compilation. For example, assuming names is an array of String[] as we saw in the ﬁrst example, the following two loops are equivalent:

for(String name : names) {

System.out.print(name + ", ");

}

for(int i=0; i < names.length; i++) {

String name = names[i];

System.out.print(name + ", ");

}

For objects that inherit java.lang.Iterable, there is a different, but similar, conversion. For example, assuming values is an instance of List<Integer>, as we saw in the second example, the following two loops are equivalent:

for(int value : values) {

System.out.print(value + ", ");

}

for(java.util.Iterator<Integer> i = values.iterator(); i.hasNext(); ) {

int value = i.next();

System.out.print(value + ", ");

}

Notice that in the second version, there is no update statement as it is not required when

using the java.util.Iterator class.

An object is a runtime instance of a class in memory. All the various objects of all the different classes represent the state of your program.

Java classes have two primary elements: **methods**, often called functions or procedures in other languages, and **fields**, more generally known as variables. Together these are called the **members** of the class. Variables hold the state of the program, and methods operate on that state. If the change is important to remember, a variable store that change.

Java building blocks:

* class
* interface
* enum
* annotation

Java calls a word with special meaning a keyword

Method requires information be supplied to it from the calling method, this information is called a parameter. But concrete parameter value is an argument

Method signature – method name and parameters (parameters order plays a role, namely if parameters types are different than different order of them result in different signature)

Comments aren’t executable code and could be place anywhere

Spaces between / and \* (or \*\*) in multiline comment cause compile error

**Classes vs. Files**

* It is possible to put several classes in single \*.java file (not common practice, but anyway).
* And such files aren't mandatory to be a public
* To compile them use javac with common java file
* To run use java with class name (it means that param for javac is class name, not class-file name)
* It is possible to have at most one public class in \*.java file. And in such a case class name and java file name should be the same. But public class couldn’t be presented at all.

Bytecode consists of (~256 that is why byte) instructions that the JVM knows how to execute. Notice that we must omit the .class extension to run Zoo.java because the period has a reserved meaning in the JVM.

The keyword static binds a method to its class, so it can be called by just the class name. Java doesn’t need to create an object to call the main() method, in fact, the JVM does this when loading the class name given to it.

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 non-static main() method might as well be invisible from the point of view of the JVM.

Java executables – classes which run using ‘java className’

The keyword **void** represents the return type. A method that returns no data returns control to the caller silently. In general, it’s good practice to use void for methods that change an object’s state. In that sense, the main() method changes the program state from started to finished

main() method’s parameter list, represented as an array of java.lang.String objects. In practice, you can write String[] args, String args[] or String... args; the compiler accepts any of these. The variable name args hints that this list contains values that were read in (arguments) when the JVM started. You can use any name you like, though.

There are two key points... are called varargs (variable argument lists)

Spaces are used to separate the arguments for java command. If you want spaces inside an argument, you need to use quotes.

All command-line arguments are treated as String objects, even if they represent another data type.

Import statements tell Java which packages to look in for classes. There is no any loading (like include in C++), only identification of the places to search in.

Package names are hierarchical.

Classes in the same package are often imported together. You can use a \* shortcut to import all the classes in a package.

The \* is a wildcard that matches all classes in the package. It doesn’t import child packages, fields, or methods; it imports only classes (unless static).

**Wildcard import overheads**

You might think that including so many classes slow down your program, but it doesn’t. The compiler figures out what’s needed. Which approach you choose is personal preference.

There’s one special package in the Java world called java.lang. This package is special in that it is automatically imported.

Classes without explicit stated package automatically put in default (no-name) one. But classes from such package couldn’t be imported in any explicitly named package

Another case of redundancy involves importing a class that is in the same package as the

class importing it. Java automatically looks in the current package for other classes.

Now let’s consider some imports that don’t work:

import java.nio.\*; // NO GOOD – a wildcard only matches

//class names, not "file.\*Files"

import java.nio.\*.\*; // NO GOOD – you can only have one wildcard

//and it must be at the end

import java.nio.files.Paths.\*; // NO GOOD – you cannot import methods

//only class names

One of the reasons for using packages is so that class names don’t have to be unique across all of Java. This means you’ll sometimes want to import a class that can be found in multiple places.

import java.util.Date;. The tricky cases come about when other imports are present:

import java.util.\*;

import java.sql.\*; // DOES NOT COMPILE

When the class is found in multiple packages, Java gives you the compiler error: The type Date is ambiguous

Is it a compile error or warning?

import java.util.Date;

import java.sql.Date;

Java is smart enough to detect that this code is no good. As a programmer, you’ve claimed to explicitly want the default to be both the java.util.Date and java.sql.Date implementations. Because there can’t be two defaults, the compiler tells you: The import java.sql.Date collides with another import statement

**If You Really Need to Use Two Classes with the Same Name…**

Sometimes you really do want to use Date from two different packages. When this happens, you can pick one to use in the import and use the other’s fully qualiﬁed class name (the package name, a dot, and the class name) to specify that it’s special. For example:

import java.util.Date;

public class Conflicts {

Date date;

java.sql.Date sqlDate;

}

Or you could have neither with an import and always use the fully qualiﬁ ed class name:

public class Conflicts {

java.util.Date date;

java.sql.Date sqlDate;

}

First you declare the type that you’ll be creating (Random) and give the variable a name (r). This gives Java a place to store a reference to the object. Then you write new Random() to actually create the object in heap memory.

There are two key points to note about the constructor:

* the name of the constructor matches the name of the class
* there’s no return type.

The purpose of a constructor is to initialize fields, although you can put any code in there. Another way to initialize fields is to do so directly on the line on which they’re declared.

It’s possible to read and write instance variables directly from the caller. In this example, a mother swan lays eggs:

public class Swan {

int numberEggs;// instance variable

public static void main(String[] args) {

Swan mother = new Swan();

mother.numberEggs = 1; // set variable

System.out.println(mother.numberEggs); // read variable

}

}

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

public class Name {

String first = "Theodore";

String last = "Moose";

String full = first + last}

When you learned about methods, you saw braces ({}). The code between the braces is called a code block. Sometimes this code is called being inside the braces. Anywhere you see braces is a code block.

Sometimes code blocks are inside a method. These are run when the method is called. Other times, code blocks appear outside a method. These are called instance initializers. Also, there is a static initializer.

**Order of Initialization that you need to remember:**

* Fields and instance initializer blocks are run in the order in which they appear in the file
* The constructor runs after all fields and instance initializer blocks have run.

Order matters for the fields and blocks of code. You can’t refer to a variable before it has

been initialized:

{ System.out.println(name); } // DOES NOT COMPILE

private String name = "Fluffy";

**Primitive Types:**

Java has eight built-in data types, referred to as the Java primitive types. These eight data types represent the building blocks for Java objects, because all Java objects are just a complex collection of these primitive data types.

float and double are used for floating-point (decimal) values.

* A float requires the letter f following the number so Java knows it is a float
* byte, short, int, and long are used for numbers without decimal points
* Each numeric type uses twice as many bits as the smaller similar type. For example,

short uses twice as many bits as byte does.

You should know that a byte can hold a value from –128 to 127. So, you aren’t stuck memorizing this, let’s look at how Java gets that. A byte is 8 bits. A bit has two possible values. (These are basic computer science definitions that you should memorize)

28 is 2 × 2 = 4 × 2 = 8 × 2 = 16 × 2 = 32 × 2 = 64 × 2 = 128 × 2 = 256.

Since 0 needs to be included in the range, Java takes it away from the positive side.

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

int num;

**What Is the Largest int?**

You do not have to know this for the exam, but the maximum number an int can hold is 2,147,483,647. How do we know this? One way is to have Java tell us:

System.out.println(Integer.MAX\_VALUE);

The other way is with math. An int is 32 bits. 232 is 4,294,967,296. Divide that by 2 and you get 2,147,483,648. Then subtract 1 as we did with bytes and you get 2,147,483,647. It’s easier to just ask Java to print the value, isn’t it?

There are a few more things you should know about numeric primitives. When a number is present in the code, it is called a literal. By default, Java assumes you are defining an int value with an integer literal.

long max = 3123456789; // DOES NOT COMPILE

Java complains the number is out of range. And it is – for an int. However, we don’t have an int. The solution is to add the character **L** to the number:

long max = 3123456789**L**; // now Java knows it is a long

The lowercase l looks like the number 1.

Another way to specify numbers is to change the “base.” Java allows you to specify digits in several other formats besides 10-based system:

* octal (digits 0–7), which uses the number **0** as a prefix – for example, **0**17
* hexadecimal (digits 0–9 and letters A–F), which uses the number **0** followed by **x** or **X** as a prefix—for example, **0x**FF
* binary (digits 0–1), which uses the number **0** followed by **b** or **B** as a prefix—for example, **0b**10

The last thing you need to know about numeric literals is a feature added in Java 7. You can have underscores in numbers to make them easier to read:

int million1 = 1000000;

int million2 = 1\_000\_000;

We’d rather be reading the latter one because the zeroes don’t run together. You can add underscores anywhere except at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point. Also, for floating point numbers underscore could be used after leading zero which will be truncated. Doesn’t work with integers.

**Reference Types:**

A reference type refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to. Instead, a reference “points” to an object by storing the memory address where the object is located, a concept referred to as a pointer. Unlike other languages, Java does not allow you to learn what the physical memory address is. You can only use the reference to refer to the object.

java.util.Date today;

String greeting;

The today variable is a reference of type Date and can only point to a Date object. The greeting variable is a reference that can only point to a String object. A value is assigned to a reference in one of two ways:

* a reference can be assigned to another object of the same type.
* a reference can be assigned to a new object using the new keyword.

The Reference type objects do not have names and can be accessed only via their corresponding reference. Declared type of ref variable denotes an object (or subclass of it) on which such a link can referred to. Size of ref vars have the same size (because they are just pointer to memory where objects located. These objects in turn could vary a lot by the size)

**Key Differences between reference and primitive variables:**

There are a few important differences you should know:

1. reference types can be assigned null, which means they do not currently refer to an object. Primitive types will give you a compiler error if you attempt to assign them null
2. reference types can be used to call methods when they do not point to null. Primitives do not have methods declared on them
3. all the primitive types have lowercase type names. All classes that come with Java begin with uppercase. You should follow this convention for classes you create as well

A variable is a name for a piece of memory that stores data. When you declare a variable, you need to state the variable type along with giving it a name.

**Declaring Multiple Variables:**

You can also declare and initialize multiple variables in the same statement. The shortcut to declare multiple variables in the same statement only works when they share a type, i.e. code won’t compile we try to declare multiple variables of different types in the same statement.

**Initializing Instance Members**

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

{

// whatever code is needed for initialization goes here

}

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

static init block is called every time class is loading (declared such class, or call static members etc.)

Top level class in Java:

* <https://stackoverflow.com/questions/7370808/why-cant-a-top-level-class-be-static-in-java>

static modifier:

* <https://javarush.ru/groups/posts/800-10-zametok-o-modifikatore-static-v-java>
* <http://citforum.ru/internet/javaqa/javaqanda_14.shtml>

Variable declaration and initialization can be put on one line. However, this form of initialization has limitations because of its simplicity. If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate

It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

**There are only three rules to remember for legal identifiers:**

* The name must begin with a letter or the symbol $ (or any currency symbol) or \_
* Subsequent characters may also be numbers
* You cannot use the same name as a Java reserved word. As you might imagine, a reserved word is a keyword that Java has reserved so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don’t, though

**An object is no longer reachable when one of two situations occurs:**

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

Luckily, there isn’t much to remember about finalize() for the exam. Just keep in mind that it might not get called and that it won’t be called twice. With that said, this call produces no output when it run.

A Java operator is a special symbol that can be applied to a set of variables, values, or literals – referred to as operands – and that returns a result. Three flavors of operators are available in Java: unary, binary, and ternary. These types of operators can be applied to one, two, or three operands, respectively.

int y = 4;

double x = 3 + 2 \* --y;

In this example, you would first decrement y to 3, and then multiply the resulting value by 2, and finally add 3. The value would then be automatically up cast from 9 to 9.0 and assigned to x. The final values of x and y would be 9.0 and 3, respectively.

**Fails when you are trying to convert from larger to smaller data types:**

Let’s consider some examples to show how casting can resolve these issues

int x = 1.0; // DOES NOT COMPILE

short y = 1921222; // DOES NOT COMPILE

int z = 9f; // DOES NOT COMPILE

long t = 192301398193810323; // DOES NOT COMPILE

* The 1st statement does not compile because you are trying to assign a double 1.0 to an

integer value. Even though the value is a mathematic integer, by adding .0, you’re instructing the compiler to treat it as a double

* The 2nd statement does not compile because the literal value 1921222 is outside the range of short and the compiler detects this
* The 3rd statement does not compile because of the **f** added to the end of the number that instructs the compiler to treat the number as floating-point value
* The 4th statement does not compile because Java interprets the integer literal as an int and notices that the value is larger than int allows. The literal would need a postfix **L** to be considered a long.

Direct/indirect codes:

* <https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D1%8F%D0%BC%D0%BE%D0%B9_%D0%BA%D0%BE%D0%B4>
* <https://math.semestr.ru/inf/inverse.php>

Numerical systems:

* <https://math.semestr.ru/inf/index.php>

One final thing to know about the assignment operator is that the result of the assignment is an expression in and of itself, equal to the value of the assignment. For example, the following snippet of code is perfectly valid, if not a little odd looking:

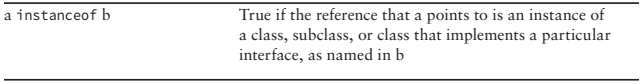
long x = 5;

long y = (x=3);

System.out.println(x); // Outputs 3

System.out.println(y); // Also, outputs 3

Note: should work even without ()



A Java statement is a complete unit of execution in Java, terminated with a semicolon (;)

may notice the syntax of the continue statement mirrors 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 and ignore any indentation you may come across.

The **else** operator takes a statement or block of statement, in the same manner as the **if** statement does. In this manner, we can append additional **if-then** 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-then** statement that evaluates to true. If neither of the first two expressions are true, it will execute the final code of the else block.

**Ternary Expression Evaluation:**

As of Java 7, only one of the right-hand expressions of the ternary operator will be evaluated at runtime. In a manner like the 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. Let’s illustrate this principle with the following example:

int y = 1;

int z = 1;

final int x = y<10 ? y++ : z++;

System.out.println(y+","+z); // Outputs 2,1

Notice that since the left-hand boolean expression was true, only y was incremented. Contrast the preceding example with the following modiﬁcation:

int y = 1;

int z = 1;

final int x = y>=10 ? y++ : z++;

System.out.println(y+","+z); // Outputs 1,2

Now that the left-hand boolean expression evaluates to false, only z was incremented. In this manner, we see how the expressions in a ternary operator may not be applied if the expression is not used. For 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.

**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. This means you can use only literals, enum constants, or final constant variables of the same data type. **By final constant**, we mean that the variable must be marked with the final modifier and **initialized** with a literal value **in the same expression** in which it is declared.

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 that is proceeded by a colon (:).

Labels are extremely useful in nested environments. Optional labels are often only used in loop structures. While this topic is not on the OCA exam, it is possible to add optional labels to control and block structures. That said, it is rarely considered good coding practice to do so.

For formatting, labels follow the same rules for identifiers. For readability, they are commonly expressed in uppercase, with underscores between words, to distinguish them from regular variables.

You may notice the syntax of the **continue** statement mirrors that of the **break** statement. In fact, the statements are similar 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.

Decision making in Java:

* <https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/amp/>

Without examining whether you should, yes, you can use **labeled statements with** **if** in Java. According to the 1.7 specification: The Identifier is declared to be the label of the immediately contained Statement. [...] identifier statement labels are used with break (§14.15) or continue (§14.16) statements appearing anywhere within the labeled statement.

It goes on (emphasis added)

If the statement is labeled by an Identifier and the contained Statement completes abruptly because of a break with the same Identifier, then the labeled statement completes normally. In all other cases of abrupt completion of the Statement, the labeled statement completes abruptly for the same reason.

So, if you break an **if** block (remember a block is a statement), you can exit the if body. Let's test it:

public static void main(String[] args) {

if (true) label: {

if (args != null)

break label;

System.out.println("doesn't get here");}

System.out.println("Outside of labeled block");

}

Output:

Outside of labeled block

Placing one String before the other String and combining them together is called string concatenation. The OCA exam creators like string concatenation because the + operator can be used in two ways within the same line of code (operator overloading). There aren’t a lot of rules to know for this, but you must know them well:

* If both operands are numeric, + means numeric addition
* If either operand is a String, + means concatenation
* The expression is evaluated left to right.

immutable classes in Java are ﬁnal, and subclasses can’t add mutable behavior.

**The String Pool:**

Since strings are everywhere in Java, they use up a lot of memory. In some production applications, they can use up 25–40 percent of the memory in the entire program. Java realizes that many strings repeat in the program and solves this issue by reusing common ones. The **string pool**, also known as the **intern pool**, is a location in the Java virtual machine (JVM) that collects all these strings. The string pool contains literal values that appear in your program. For example, “name” is a literal and therefore goes into the string pool. myObject.toString() is a string

but not a literal, so it does not go into the string pool. Strings not in the string pool are garbage collected just like any other object. Remember back when we said these two lines are subtly different?

String name = "Fluffy";

String name = new String("Fluffy");

The former says to use the string pool normally. The second says “No, JVM. I really don’t want you to use the string pool. Please create a new object for me even though it is less efficient.” When you write programs, you wouldn’t want to do this. For the exam, you need to know that it is allowed.

**Size vs. Capacity:**

The behind-the-scenes process of how objects are stored isn’t on the exam, but some knowledge of this process may help you better understand and remember String-Builder. Size is the number of characters currently in the sequence, and capacity is the number of characters the sequence can currently hold. Since a String is immutable, the size and capacity are the same. The number of characters appearing in the String is both the size and capacity. For StringBuilder, Java knows the size is likely to change as the object is used. When StringBuilder is constructed, it may start at the default capacity (which happens to be 16) or one of the programmer’s choosing.

Algorithms and data structures:

* <https://proglib.io/p/algorithms-and-structures/>

LocalTime.isBefore() can’t compare a LocalDate object to a LocalTime object – compilation fails.

SJCP Exam Preparation: Top-level and Inner Classes:

<https://www.developer.com/java/ent/article.php/859381/sjcp-exam-preparation-top-level-and-inner-classes.htm>