1 What is Java?

Java is intended to be a platform-independent, object-oriented programming language. The idea behind Java is to have a “standard” platform, where you write code just once, and it can be run on any operating system with no modification. This is no simple task, since there are different operating systems running on different hardware architectures. Thus, the Java platform is a little more complicated than “native” programming languages.

1.1 The Java Compiler

Normally, you would expect a compiler (say, a C or Pascal compiler) to take source code as input, and produce executable (well, at least linkable) object code as output. The Java compiler does not do that. It takes Java source code as input (files with .java extension) and produces Java object files (files with .class extension). Java object files are in what is called *bytecode*. It can not be executed directly on any of the popular operating systems. So how do you execute Java bytecode? That is the job of the Java Virtual Machine.

1.2 The Java Virtual Machine

The Java Virtual Machine is a program that runs on top of the host operating system and *emulates* a machine whose native code is Java bytecode. In other words, it takes Java bytecode, and converts it into the language of the host operating system, and then executes it.

In fact, this is exactly how the platform-independence of Java works. Someone has already written the code to convert Java bytecode into the native language of all popular operating systems and hardware architectures. So, that “translation” layer is transparent for you, the application programmer.

1.3 Strenghts and Weaknesses of Java

The main and obvious strength of Java is its platform-independence. That alone makes it a very useful tool for web applications. The syntax of the Java language is very similar to that of C and C++, which makes the trasition very easy for people who are familiar with the C/C++ syntax.

There are also additional advantages due to properties of the language. Java is not a scripting language. It is also strongly typed, so at compile time your code is strictly verified by the compiler. In essence, a Java program which does compile without errors (There are no warnings in Java. It is an error, or it is not.) is more likely to run properly than in any other progamming language (those that I know, at the least).

The main weakness of Java is also its platform-independence. While the intention is to have a “write once, run everywhere” language, in practice Java sometimes becomes a “write once, *debug* everywhere” language. This is because the Java Virtual Machine relies on the host operating system for most input/output operations (graphical user interface, networking) and each operating system has its own way of doing things and its own set of bugs. So, while the code is portable in principle, it may need serious testing on every platform it is intended to be run.

In addition, while the memory management of Java is much easier compared to any conventional programming languages, it is a memory hog. When a Java Virtual Machine is run, it will consume anywhere from 40MB to infinity of RAM. Moreover, there is also a performance hit, since the code first needs to be converted to native code. This performance hit is not as bad as it used to be, since these days most Java Virtual Machines use JIT (Just In Time) compilation. This means that the Java bytecode is converted to native code before execution, and then runs (mainly) as native code. However, Java is still not a suitable platform for computation-intensive programming.

2 Introduction to Object-Oriented Programming

Java is by nature, an object-oriented language. It does not only *allow* object-oriented programming, it in fact *forces* the programmer to use object-oriented techniques, as far as a programming language can. This is in contrast with C++. C++ *supports* the use of object-oriented code, but you can still write programs without using any classes or objects.

2.1 What are Objects?

Generally speaking, everything can be thought of as an object. A person, a hat, a tree, a room, a computer and an apple can all be thought of as objects.

Every object has its own set of properties, and actions it can perform or can be performed on it. These would be the *data members* and *methods* of an object in object-oriented speak.

Thinking of apples, there are many apples in existence. Some examples could be myApple, yourApple, theRedApple. There are also computers in existence, for example myComputer, oldComputer, and hisLaptop. These are all objects. But how are the apples different than the computers? They belong to the same *class* of objects. Computers belong to a different *class* of objects.

2.2 Object Oriented Programming

Object oriented programming is different from the more traditional, procedural programming in the following sense: Although in both cases there are procedures and there are "objects" (even structs in C can be considered to be objects), the "primary" role is given to objects in object-oriented programming, while it is given to procedures (or functions) in procedural programming.

Object oriented programming is not the solution to every problem. For instance, for short, computational-only problems, the odds are using object-oriented methods will cause more overhead than the help it will provide. The virtues of object orientation are explicitly visible only in large projects, which are meant to be maintained over time.

3 The "Hello, world!" Program in Java

It is customary to start programming in any programming language with a "Hello, world!" program, which does nothing except printing things out. It will demonstrate the basic structure of Java programs.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

In order to run this program, you need to type it exactly as shown, into a file called "HelloWorld.java". Java source code files always have the extension ".java", and the files compiled into bytecode have the extension ".class". So, when this code is compiled, the compiler will generate a file called "HelloWorld.class".

Normally, in Java, each class resides in a file that carries the same name as the class. There will be one or more classes in every Java program. However, execution must begin *somewhere*. The place program execution begins is the `main` method (*method*s are functions belonging to a class in Java) of one class. One and only one class in the whole program may contain a `main` method, and it *must* be given exactly as follows:

```
public static void main(String[] args) {
    <method body>
}
```

The first line is the *declaration* line of the method, while the portion enclosed in braces is the *body of the method*, containing statements telling the computer what to do.

The part enclosing the `main` method is a class definition. It defines the class called `HelloWorld`. Again, the portion contained in braces is the body of the class. For now, you can consider these "magic incantations", since we will first learn what we can put inside the `main` method before going on to anything else.

Last, but not least, the line

```
        System.out.println("Hello, world!");
```

tells the computer to print the string "Hello, world!" on a line by itself. Again, for now consider this magic; if you need to print anything, you type a statement similar to this. Another important point is that Java is case-sensitive, so you must type things in the correct case.

4 Variable Types in Java

Java has a few basic variable types that are not objects. Note that Java is a *strongly typed* language, and all variables must be declared with their proper types.

The integer and floating-point variable types in Java are given in the following two tables:

Variable Type	Size	Minimum Value	Maximum Value
byte	1 byte	-128	127
short	2 bytes	-32768	32767
int	4 bytes	-2147483648	2147483647
long	8 bytes	-9223372036854775808	9223372036854775807

Variable Type	Size	Minimum Value	Maximum Value
float	4 byte	-3.4E38	3.4E38
double	8 bytes	-1.7E308	1.7E308

5 Variable Declarations

The general form of variable declaration in Java is:

```
<variable-type> <identifier> [= <initial value>];
```

Examples:

```
long bigInteger;
short myWeight;
int counter = 0;
```

Note that it is possible to declare and initialize more than one variable with a single declaraton, using commas. Example:

```
float sideOne = 3.0, sideTwo = 4.0, sideThree = 5.0;
```

6 Integer and Floating-Point Literals

A literal is any value that is entered literally in the Java source code, such as 517, 1E12 or -2211. Any literal integer value is by default of type int. If you need to use a literal value larger than the range allowed by the int data type, you need to append the letter L to the end of the number. You can also use a lower case l, however it can easily be mixed up with 1.

Any floating point literals are of type double by default. If you want the literal to be of type float, you need to append the letter F to end of the number.

You can also use the exponential notation when using a floating point literal. An example would be 6.02E23, which means $6.02 \cdot 10^{23}$.

7 Assignment and Arithmetic Calculations

The assignment operator in Java is the plain equals sign "=". For example:

```
int numApples = 5;
int numOranges = 10;
int numFruit = 0;

numFruit = numApples + numOranges;
```

The last line assigns the sum of numApples and numOranges to numFruit, so the value of numFruit will be 15. This also introduces the use of the plus sign "+" for addition. The four basic arithmetic operators, "+", "-", "*", "/" can be used in both floating-point and integer calculations.

When the operands in an arithmetic calculation are of different types, one operand is "promoted" to match the other. The promotion order is as follows:

$$\text{byte} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$$

When both operands of the division operator "/" are integers, the result will also be an integer. For example:

```
12/4 = 3
5/3 = 1
12/5 = 2
```

In integral division, it is an error to divide by zero, and an exception will be thrown.

Division by zero when the operands are floating point numbers will not result in an error. However, the result will be infinity, and trying to print the value of the variable will yield "Inf". Infinity will act as one expects in calculations. When a calculation does not have a defined result (such as 0/0 or infinity/infinity) the result will be "NaN", which stands for Not a Number.

8 Casting

When you need to explicitly change the type of an expression, you need to use explicit casting. For example, look at the code fragment below:

```
double result;
int three = 3;
int two = 2;

result = 1.5 + three/two;
```

At the end of the calculation, result will have the value 2.5. This is because when "three" is divided by two, it is done by integer division, and the result is 1. When you add 1.5 and 1, 1 is promoted to a double (remember, the 1.5 is a double by default) and added to 1.5, resulting in 2.5.

This, apparently is not what is intended in the code. We can change the behaviour by explicitly casting the value of the variable three into a double as follows:

```
double result;
int three = 3;
int two = 2;

result = 1.5 + (double)three/two;
```

In this case, "three" is converted to a double, and then the division is performed. Since now one operand is a double, the other is also promoted to a double, and at the end, result will have the value 3.0, as one would expect reading the code.

The general form of casting is as follows:

```
(<type-name>)expression
```

9 More Operators

Apart from the well-known arithmetic operations, there is also the modulus operator, %. The result is the remainder when the first operand is divided by the second operand. For example:

```
9 % 2 = 1
11 % 3 = 2
```

Interestingly, the % operator works just as well for floating-point operands. The result is again, the remainder when the first operand is divided by the second operand an integral number of times. For example:

```
4.5 % 2.0 = 0.5
2.2 % 1.0 = 0.2
```

There is also another set of operators, which are of the "op=" type. For each operator we have seen, there is a corresponding "op=" operator. Their meaning is best explained by examples. In the examples below, each line with an "op=" operator is equivalent to the preceding statement without an "op=" operator.

```
a = a + 5;
a += 5;

b = b / 2.0;
b /= 2.0;

c = c * 2;
c *= 2;
```

Finally, there are the increment and decrement operators, which have both a postfix and prefix form. The increment operator is $++$ and the decrement operator is $--$. When placed before or after a variable, they increment or decrement the value of that variable by 1 (1.0 if it is a floating point variable). The postfix version does the increment/decrement operation *after* the current expression is fully evaluated, while the prefix version does it *after* the current expression is evaluated.

10 Bitwise Operators

Bitwise operators operate on the binary digits of the numbers. The bitwise operators are as follows:

Operator	Operation
&	AND
|	OR
~	NOT
^	XOR

Java also has three bitwise-shift operators:

Operator	Operation
<<	Shift left, fill with zeros
>>	Shift right, propagate sign bit
>>>	Shift right, fill with zeros

11 More Variable Types

There are two more variable types in Java, “char” and “boolean”. The “char” type can hold one unicode character, while the boolean variable type can take values of true and false only. Here are examples of their use:

```
char myCharacter = 'Y';
boolean isOpen = false;
```

12 An Example: Simple Calculation and Printing

```
public class AddAndPrint {
    public static void main(String[] args) {
        int a = 4;
        int b = 5;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("a+b = " + (a+b) + ".");
    }
}
```