Lecture - Getting Started with Java

Overview

In this module, you will learn:

- What is the structure of a Java program?
- How to write, compile, and run a Java program?
  - Using a simple text editor (e.g., Vim)
  - Using an Integrated Development Environment (IDE) (e.g., Eclipse)
- Variable declaration & assignment

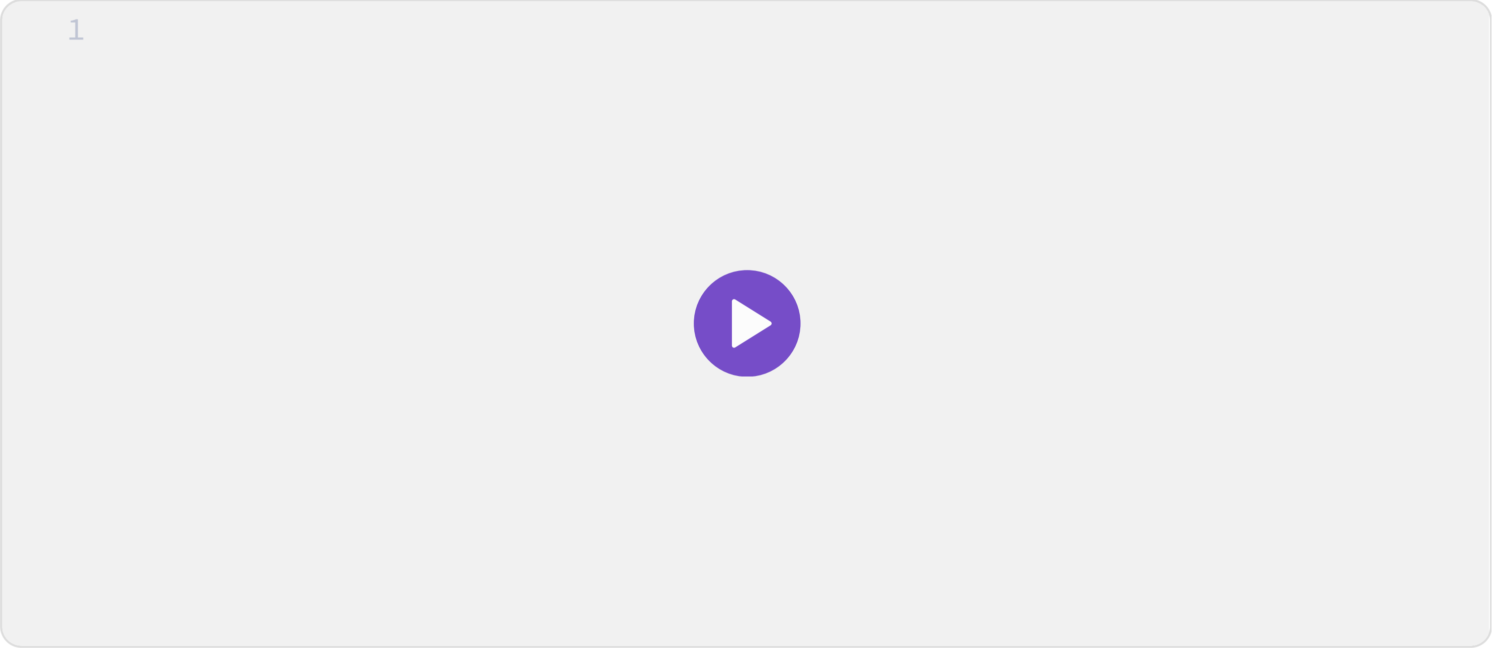
Java - A Brief History

Java is a general-purpose programming language whose development goes back as far as 1991, where James Gosling led a team at Sun Microsystems to design a language for programming home appliances, such as dishwashers and television sets. The diverse nature of these appliances required the designers of Java to create programs that worked on different processors. To avoid having to write custom compilers for each platform, the team devised a two-step translation process: 1) program code would first be translated into an **intermediate language (byte-code)**, which is universal for all platforms and 2) a simple and inexpensive program (i.e., the interpreter) would translate this intermediate language into the machine language for the particular platform it should run on.

While the language didn't really take off as a go-to language for programming appliances, Patrick Naughton and Jonathan Payne used it in 1994 to run programs in a web browser and distribute programs over the Internet. Subsequently, Netscape created its web browser (Netscape Navigator) in 1995, which was capable of running Java programs.

When other browsers followed suit, Java evolved into a widely-used platform-independent programming language. In 2010, Sun Microsystems–and with it the Java language–were acquired by Oracle.

Your First Java Program



Your First Java Program

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

(See the blue "Run" button? Go head, press it!)

A Java program is made up of one or more *classes* (*Hello* in our example above). It contains zero or more *methods* (e.g., the *main* method) and instance variables (more about these later). Additionally, we have *statements*, such as:

```java
System.out.println("Hello World!");
```

Standard Output

The above statement prints the text *Hello World!* out to the console (the console is also called *terminal*).

> System.out is called "standard out" or stdout, rather than "the terminal" because the operating system can do other things with it like sending it to a file, or as input to another program.

The “ln” part means there is a *new line* character at the end of the text. Hence, any output that follows will start on a new line.

To print something without moving to another line, use

```java
System.out.print("Hello World!");
```

Of course, Java also knows comments:

```java
//a single line comment or

/*
a multi-line
comment
*/
```

These are useful for making notes to someone who reads the code later (or yourself -- we forget quickly!). They are also useful for temporarily making the system ignore some code without deleting it; this is very common during debugging.

Program Files

Every line of Java code must be in some text file. The filename must match the class name, including upper and lower casing, and always ends with .java

Our example above, therefore, should be saved in a file called *Hello.java*

Most people use an Integrated Development Environment (IDE) to create Java code. IDEs let you compile and run your code with one click and they hide some low-level details. You should, however, always make an effort to understand what happens under the hood.

Popular IDEs include Intellij, Eclipse and Netbeans, which are free to download and use. You may use whatever tool for writing code you like or runs on your system (Mac, Linux and Windows may have different options for editors here). However, **it is your responsibility to make sure assignment submissions run on ed**. Allow enough time to overcome any issues with porting from your IDE to ed.

Program Compilation and Execution

Edstem.org makes it really easy to run the examples in this class and allow you to code entire assignments. You should, however, set up your own development environment on your local computer to really understand how compilation and program invocation works. The first tutorial will tell you exactly how to set up your programming environment.

Java, just like most modern programming languages, is designed to make it easy to read and write code. Languages that are human-readable and, therefore, easy to understand, are called **high-level languages**. On the other hand, the language that the computer can directly understand, is called **machine language or a low-level language**.

A program written in a high-level language, such as Java, is translated into a program in machine language by a **compiler** through a process called **compilation**.

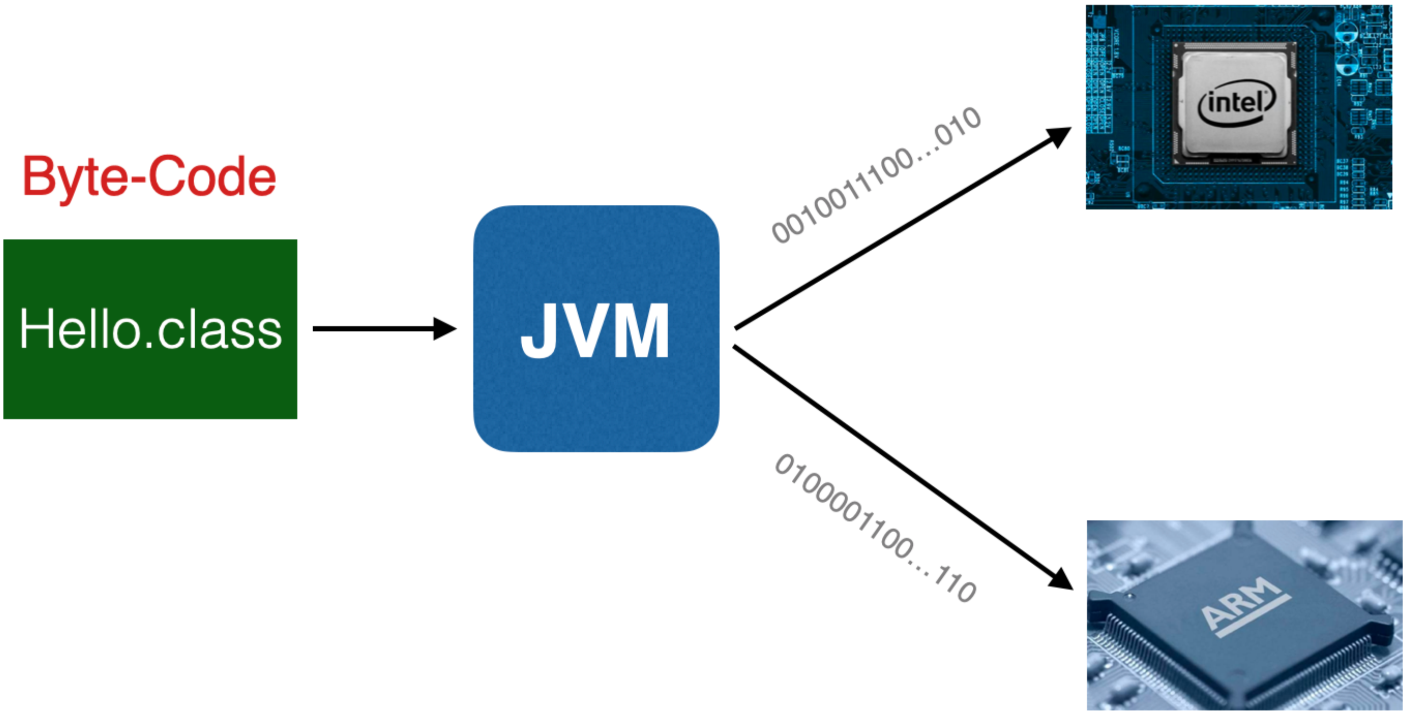
Compilation

Before a java program can be run, it must first be compiled. It checks whether your program obeys the rules of Java and produces a *.class* file (if compiler passes). The Java compiler translates your Java program (e.g., *Hello.java*) into byte-code using the *javac* command:

```
$ javac Hello.java
```

Note that the $ in the example above only indicates that the command is executed in the console. It represents the *prompt* that the computer displays when it is ready to accept a command.

Once compiled, you should see that a new file has been created: *Hello.class*. This file contains the byte-code for a fictitious computer called the Java Virtual Machine (JVM). The JVM subsequently translates a program written in byte-code into a program in the machine language for any particular computer:

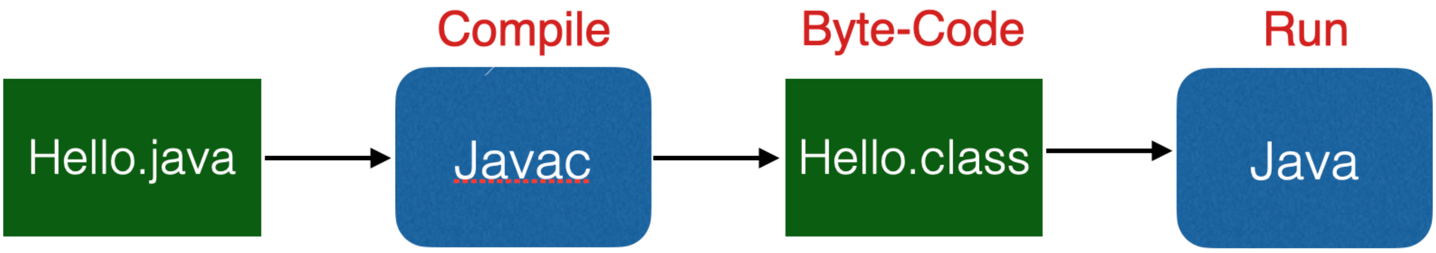


Technically, the JVM has two ways to do this translation: through an interpreter and through a Just-In-Time (JIT) compiler.

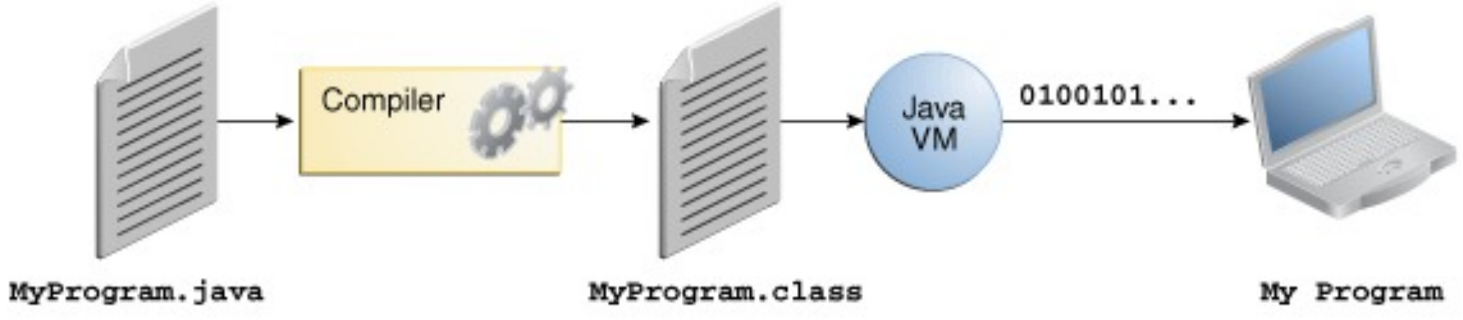
To run the compiled Java program *Hello.class*, use the JVM for your computer to translate the byte-code instructions to machine language and run the machine language instructions as follows:

```
$ java Hello
```

Java byte-code allows your Java program to be portable. Once compiled, you can use the *Hello.class* file on any computer without the need to recompile it. Hence, you can share your file with other computers, which is why Java was originally adopted for Internet applications. The following diagram summarizes the process from code compilation to execution:



It could be further illustrated as:



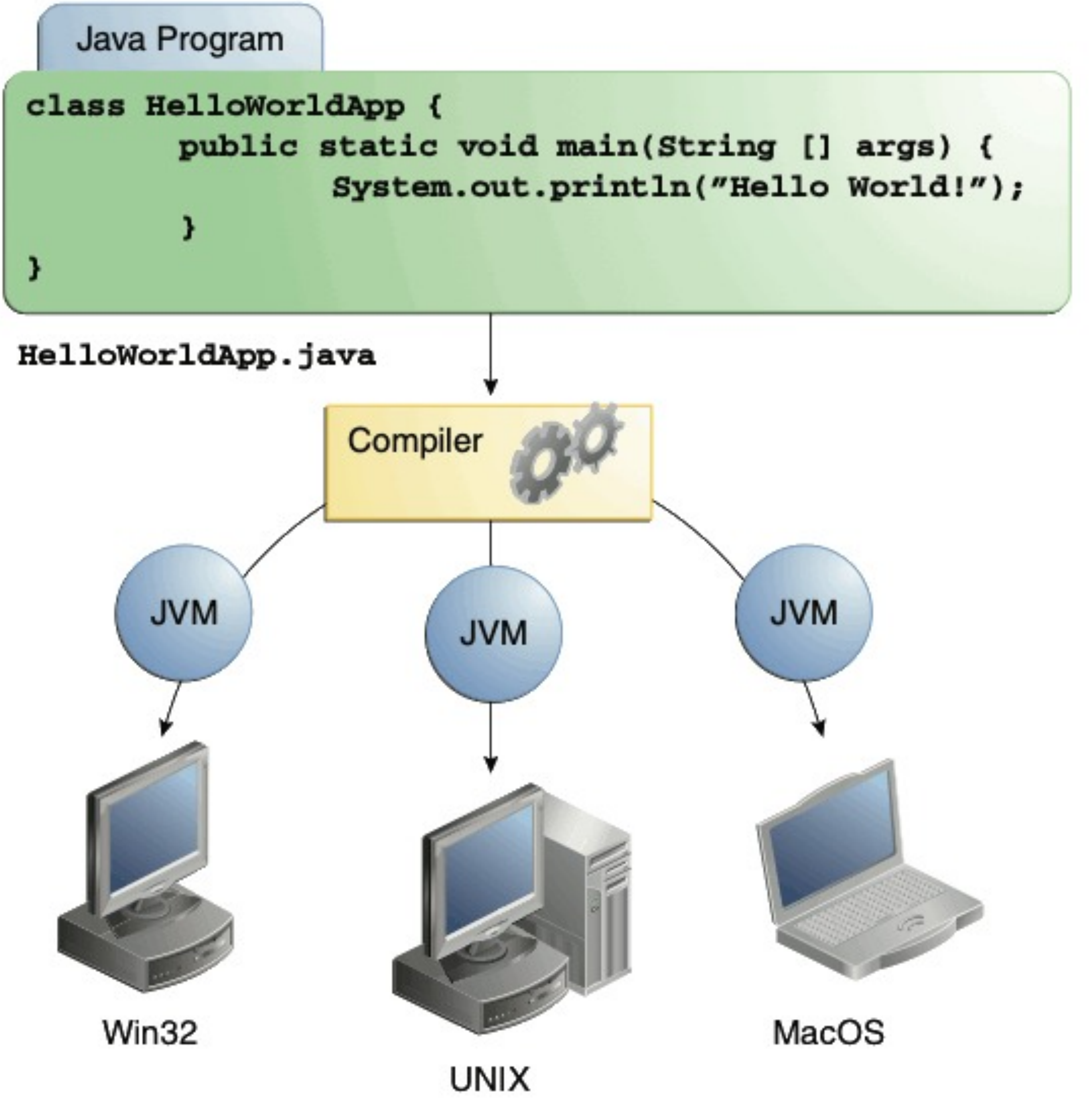
Java applications run in a console. Computer consoles used to look like this:

Most of the programs in this subject will use the console text input/output without a graphical user interface (GUI). IDEs have ways to simulate console input and output.

What is a Byte Code?

Bytecode is a highly optimised set of instructions designed to be executed by what is called the **Java Virtual Machine (JVM)**, which is part of the **Java Runtime Environment (JRE)**. In essence, the original JVM was designed as an interpreter for bytecode. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once a JRE exists for a given system, any Java program can run on it (thus achieving platform independence). Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.



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

JVM makes Java Program Secure

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it manages program execution. Thus, it is possible for the JVM to create a restricted execution environment, called the sandbox, that contains the program, preventing unrestricted access to the machine. Safety is also enhanced by certain restrictions that exist in the Java language.

What is Just-in-time (JIT)?

Java, initially designed as an interpreted language, can enhance performance by compiling bytecode into native code on-the-fly. HotSpot technology, introduced soon after Java's launch, includes a Just-In-Time (JIT) compiler for bytecode. With JIT compilation, bytecode is compiled into executable code in real time, on a piece-by-piece basis, when needed during execution. Not all bytecode is compiled—only selected portions that will benefit from compilation. The rest of the code is simply interpreted. Despite dynamic compilation, Java's portability and safety features are maintained, as the JVM remains in control of the execution environment.

What is Java Development Kit (JDK)?

It is kit that install when you install java on your system. the JDK empowers you to create Java applications from scratch, from writing the code to compiling, debugging, and finally packaging it for distribution. It's your go-to toolkit for making your Java development journey smooth and efficient. JDK includes tools such as the Java compiler (javac), Java runtime environment (java), JVM and other tools needed for Java development.

Programming with Data

It is important to state that the Java is a strongly typed language meaning that every variable, expression, and type is strictly defined. In Java, all assignments, including those in method calls, are checked for type compatibility. Unlike some other languages, Java does not allow automatic coercions or conversions of conflicting types. The Java compiler checks all expressions and parameters to ensure that the types are compatible, and any type mismatches are considered errors that must be fixed before the class can be compiled.

Data Types

Generally speaking, there are two parts to any computer program: **code** and **data**.

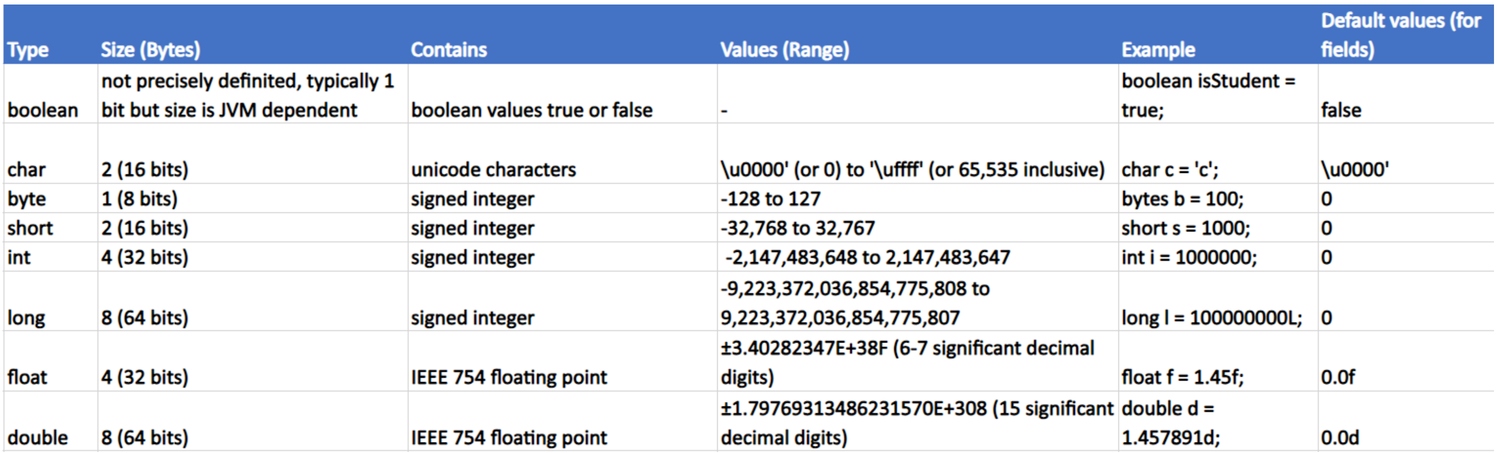
Code is the text of the program that determines what operations the program performs.

Data is what the code operates on.

Each datum (singular of data) has a type. Java distinguishes between three groups of data types: primitive, class, and array.

Primitive Types

Primitive types are the building blocks. All data are built from primitives. Primitives are atomic, which means they can’t be broken into smaller parts. The following primitive types are found in Java:



Type	Size (Bytes)	Contains	Values (Range)	Example	Default values (for fields)
boolean	not precisely definited, typically 1 bit but size is JVM dependent	boolean values true or false	-	boolean isStudent = true;	false
char	2 (16 bits)	unicode characters	\u0000' (or 0) to '\uffff' (or 65,535 inclusive)	char c = 'c';	\u0000'
byte	1 (8 bits)	signed integer	-128 to 127	bytes b = 100;	0
short	2 (16 bits)	signed integer	-32,768 to 32,767	short s = 1000;	0
int	4 (32 bits)	signed integer	-2,147,483,648 to 2,147,483,647	int i = 1000000;	0
long	8 (64 bits)	signed integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l = 100000000L;	0
float	4 (32 bits)	IEEE 754 floating point	±3.40282347E+38F (6-7 significant decimal digits)	float f = 1.45f;	0.0f
double	8 (64 bits)	IEEE 754 floating point	±1.79769313486231570E+308 (15 significant decimal digits)	double d = 1.457891d;	0.0d

These primitive types can be put into 4 groups

Integers: This group includes *byte*, *short*, *int*, and *long*, which are for whole-valued signed numbers.

Floating-point numbers: This group includes *float* and *double*, which represent numbers with fractional precision.

Characters: This group includes *char*, which represents symbols in a character set, like letters and numbers.

Boolean: This group includes *boolean*, which is a special type for representing true/false values.

A boolean, for example, can have two values: *true* and *false*. In Java, each boolean value is stored in a single byte in the computer's memory. An *int* is represented by 4 bytes, which limits its range:

$$\pm 2 \times 10^9$$

Compare this to the range of a *long* variable:

$$\pm 10^{19}$$

which takes twice as much memory. Efficient programming code uses appropriate variable types to make sure that memory is optimally used.

Integer types (byte, short, int, long) can hold any *whole number* within their range. Floating point numbers (float, double) can hold fractions as well, and have a huge range, but can't hold all integers within their range. You can think of them as holding numbers like

$$x \times 10^y$$

where x and y are both integers (positive or negative) within a certain limited range.

Let's have a look at the following example:

```java
public class SimpleCalculation {
    public static void main(String[] args) {
        System.out.println("Let's make a simple calculation:");

        int sum;
        sum = 2 + 2;
        System.out.println("2 plus 2 is " + sum);
    }
}
```

Non Primitive Types (Also called Object Types)

Other non-primitive types that java supports are called Object types that include String, Array and Class. These types group multiple primitives together with objects, and will be covered later in the subject.

Variables

In the program code above *sum* is a variable of type *int.*

Variables have names and hold data. They can have different values at different times. In Java, variable names begin with a letter, followed by letters, digits, and underscores (_).

In this subject, we will follow Java's naming convention for variable names as follows:

- Begin with lower case letter
- Follow with lower case, except
    - Capitalise the first letter of each word in your phrase, an approach called *camel case*
- Examples for variable names: *height*, *windowHeight*, *tallestWindowHeight*
- Best practice: make them descriptive, but not too long (clear abbreviations are okay)

Each variable *must be* declared, specifying its type, like in our example above:

```
int sum;
```

First comes the type (*int*), then the variable name (*sum*). The semicolon indicates the end of each statement and is mandatory.

Variables ***must be*** assigned a value before being used. This can be done in two steps like in our example:

```
int sum;
sum = 2 + 2;
```

Declaration and assignment can also happen in a single statement:

```
int sum = 4;
```

The equal sign (=) assigns the variable named *sum* the value 4.

> i Variables are called **Fields** or **Attributes** in Object Oriented jargon.

Summary

- Java programs are written in files that contain Java classes (*classname.java)*.
- Compile and run the code using the *javac* and *java* commands
- Variables hold values, they can be assigned and reassigned
- Variables must be declared, with their types (strongly typed)
- Variables must be initialised before being used

Compared to other programming languages you may be familiar with, Java is quite unforgiving when it comes to variable declarations and assignments. If you tried to assign a *boolean* value to an *int* variable, the compiler would return the following error:

```
Hello.java:5: error: incompatible types: boolean cannot be converted to int
        sum = false;
```

```
                 ^
1 error
```

This lack of compromise is also referred to Java being *strongly typed*. It makes it a great language to learn to code with as program errors are caught early.

Python: print "Hello World!"
Lisp: (princ "Hello World!")
JavaScript: alert('Hello World!')
C: puts("Hello World!");
Java: System.out.println("Hell...!");

Formatted Output

You have already seen how to print output to the console using:

```
System.out.println();
// and
System.out.print();
```

You can exert more control over string formatting using:

```
System.out.printf();
```

This so-called **format-string** is an ordinary *string*, but can contain **format specifiers** for each argument you pass in:

```
double average = 5.0;
System.out.printf("Average: %5.2f", average);
```

A format specifier begins with % and may have a number specifying how to format the next value in the list of arguments. It ends with a letter specifying the type of the value:

```
String formatString = "%X.Y";
```

The X before the decimal point specifies the minimum number of characters to be printed.

- The full number will be printed, even if it takes more characters than specified by X
- If X is omitted, the value will be printed in its minimum width
- If X is negative, the value will be left-justified, otherwise right-justified example below
- If X starts with a zero, then the number will be padded with leading 0s (good if printing for a variable-width font, where the width of a space is different from the width of a digit); otherwise it will be padded with spaces

The Y after the decimal point specifies the number of digits of the value that are printed after the decimal point. If Y is omitted, Java decides how to format it.

The following is a list of final letters that are commonly used in a format specifier:

- **d**: an integer (no fractional part) [d stands for "decimal"]
- **s**: a string (no fractional part)
- **c**: a character (no fractional part)
- **f**: a float or double
- **e**: a float or double in exponential notation

- **g**: same as %f or %e, Java chooses
- **%**: output a percentage sign (no argument)
- **n**: end the line (no argument)

Advanced: %d is used rather than %i for integers, because this naming system is so old that when it was developed, outputting in octal or hexadecimal was just as common. These can be done with %o, %h and %H. If you're curious, see what happens if you use each of these formats to print the number "10".

Advanced: What is the difference between ending the string with %n and ending it with \n?

```
public class FormatPlay {
    public static void main(String [] args) {
        String s = "string";
        double pi = 3.1415926535;
        System.out.printf("\"%s\" has %d characters %n", s, s.length());
        System.out.printf("pi to 4 places: %.4f%n", pi);
        System.out.printf("Right>>%9.4f<<", pi);
        System.out.printf(" Left >>%-9.4f<<%n", pi);
        System.out.printf("$%.2f%n", 9.99);
        System.out.printf("%06.2f%n", 9.99);
    }
}
```

in order to print " "

go to the next line

=> "string" has 6 characters

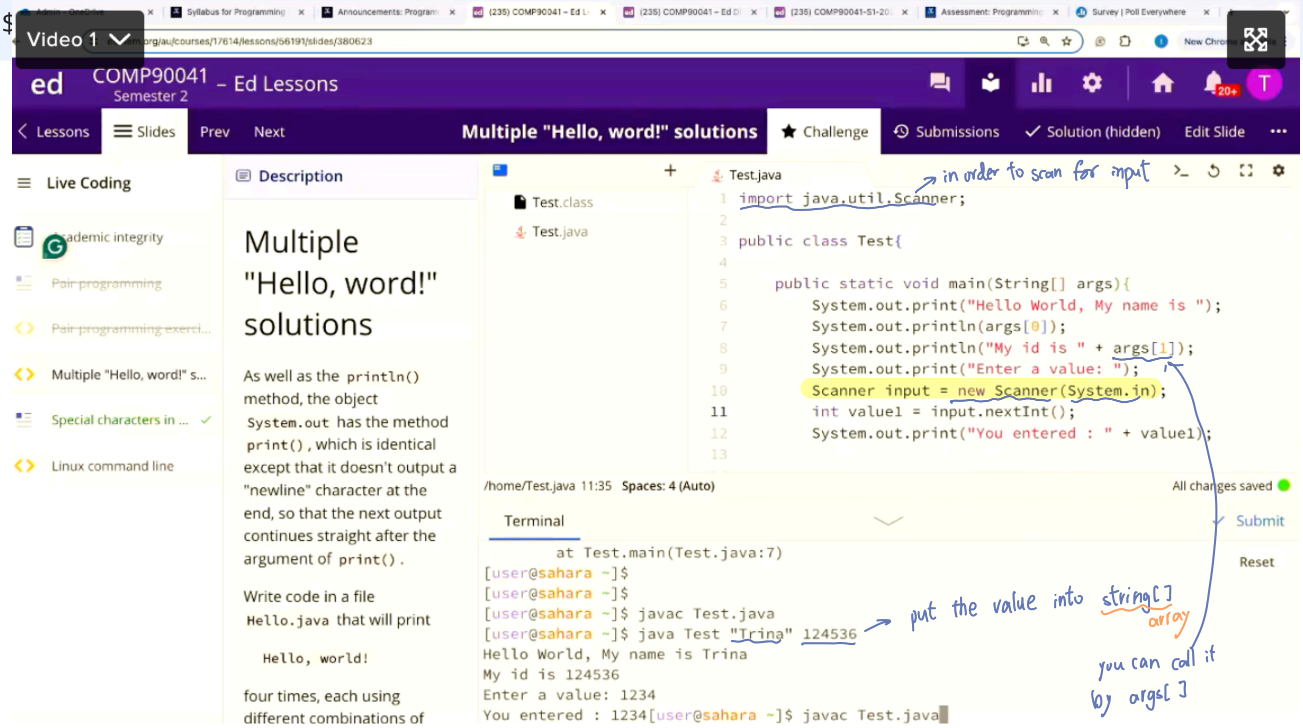
pi to 4 places: 3.1416

Right >> 3.1416<< Left>>3.1416 <<

$9.99

009.99

"$



Multiple "Hello, word!" solutions

As well as the `println()` method, the object `System.out` has the method `print()`, which is identical except that it doesn't output a "newline" character at the end, so that the next output continues straight after the argument of `print()`.

Write code in a file `Hello.java` that will print

```
Hello, world!
```

four times, each using different combinations of

```
import java.util.Scanner;

public class Test{

    public static void main(String[] args){
        System.out.print("Hello World, My name is ");
        System.out.println(args[0]);
        System.out.println("My id is " + args[1]);
        System.out.print("Enter a value: ");
        Scanner input = new Scanner(System.in);
        int value1 = input.nextInt();
        System.out.print("You entered : " + value1);

```

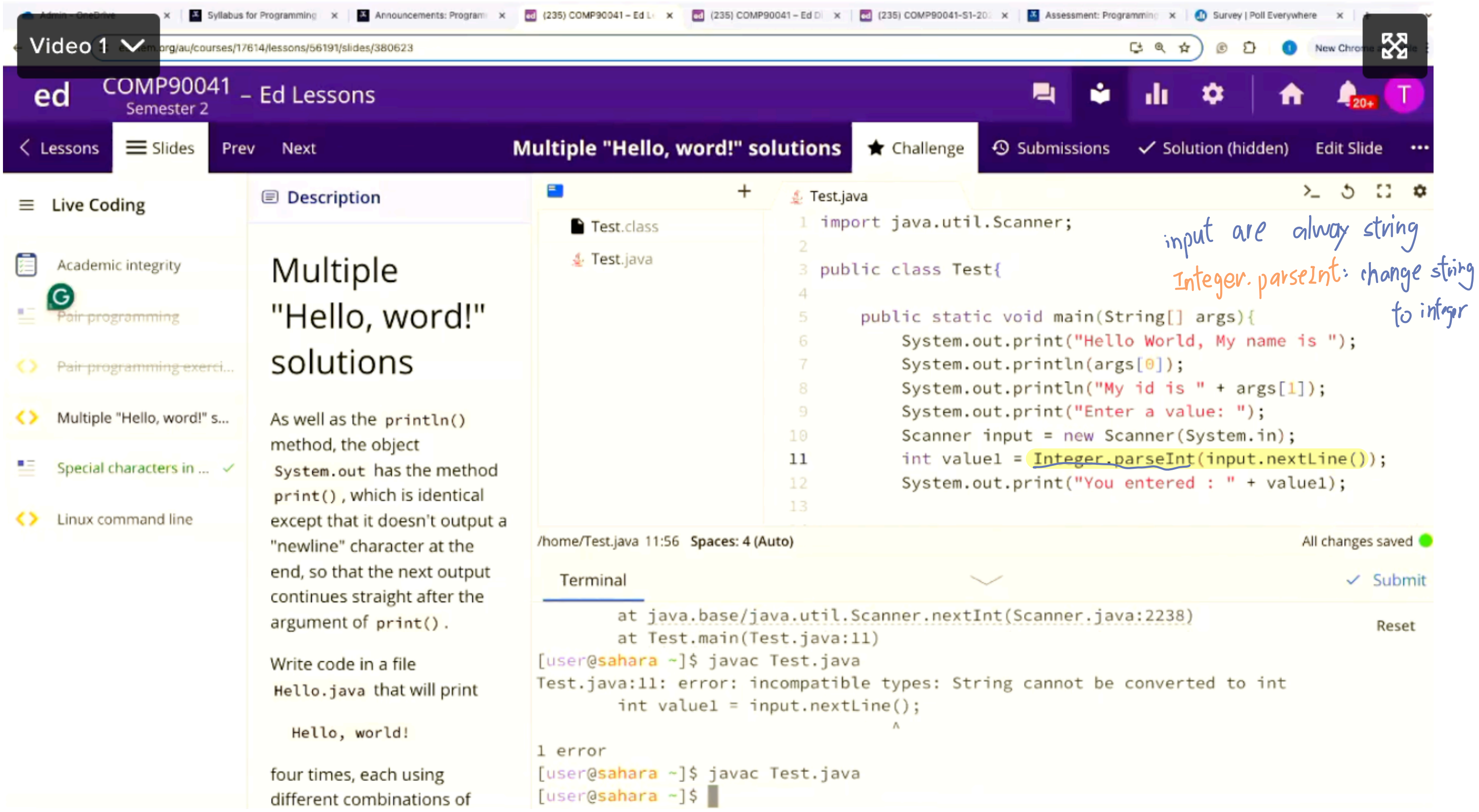
/home/Test.java 11:35 Spaces: 4 (Auto)

```
        at Test.main(Test.java:7)
[user@sahara ~]$
[user@sahara ~]$
[user@sahara ~]$ javac Test.java
[user@sahara ~]$ java Test "Trina" 124536
Hello World, My name is Trina
My id is 124536
Enter a value: 1234
You entered : 1234[user@sahara ~]$ javac Test.java
```

in order to scan for input

put the value into string[ ] array

you can call it by args[ ]



Multiple "Hello, word!" solutions

As well as the `println()` method, the object `System.out` has the method `print()`, which is identical except that it doesn't output a "newline" character at the end, so that the next output continues straight after the argument of `print()`.

Write code in a file `Hello.java` that will print

```
Hello, world!
```

four times, each using different combinations of

```java
import java.util.Scanner;

public class Test{

    public static void main(String[] args){
        System.out.print("Hello World, My name is ");
        System.out.println(args[0]);
        System.out.println("My id is " + args[1]);
        System.out.print("Enter a value: ");
        Scanner input = new Scanner(System.in);
        int value1 = Integer.parseInt(input.nextLine());
        System.out.print("You entered : " + value1);
```

```
        at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
        at Test.main(Test.java:11)
[user@sahara ~]$ javac Test.java
Test.java:11: error: incompatible types: String cannot be converted to int
        int value1 = input.nextLine();
                                   ^
1 error
[user@sahara ~]$ javac Test.java
[user@sahara ~]$ 
```

input are alway string

Integer.parseInt: change string to intger

Command Line Input

When your program is run, you can pass it arguments through the command line. When you run the following in your console, *Hello.java* is an argument for the *javac* command:

```
$ javac Hello.java
```

You can pass in additional arguments to change the program execution. This mechanism allows the user to give information to the program. Remember the *main* method, which provides the entry point for your program?

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

The variable *args* holds the values that a user provides when the program is executed.

```
public class HelloStranger {
    public static void main(String[] args) {
        System.out.println("Hello " + args[0] + "!");
    }
}
```

Go ahead and run the code snippet above on your local computer with the following program invocations:

```
$ java HelloStranger Java
Hello Java!
```

```
$ java HelloStranger
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for leng
at HelloStranger.main(HelloStranger.java:3)
```

You can add arguments to your program, which will be referred to by their index: args[0], args[1], ...

Each argument is treated as a *string*. But you can convert the *string* input to *int* or a *double*, for example using:

change string to int

```
int argument = Integer.parseInt(args[0]);
double arg0 = Double.parseDouble(args[0]);
```

change string to double

Interactive Console Input

Interactive programs allow users to provide input while they are running. To read in user input from the console, Java provides the *Scanner* class. To use it, you will need to import the class by adding the following line to the top of your program code:

```
import java.util.Scanner;
```

You need to create a *Scanner* object that is linked to your program's input stream:

```
Scanner keyboard = new Scanner(System.in);
```

variable name

Note how we have been using System.out as the program's output stream all along. System.in allows you to access its input.

Now we can use the *keyboard* object to read input from the command line as follows:

```
String line = keyboard.nextLine();
```

Let's put it all together:

```
import java.util.Scanner;

public class ScannerPlay {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Please enter your name: ");
        String name = keyboard.nextLine();
        System.out.println("Hello " + name + "!");
    }
}
```

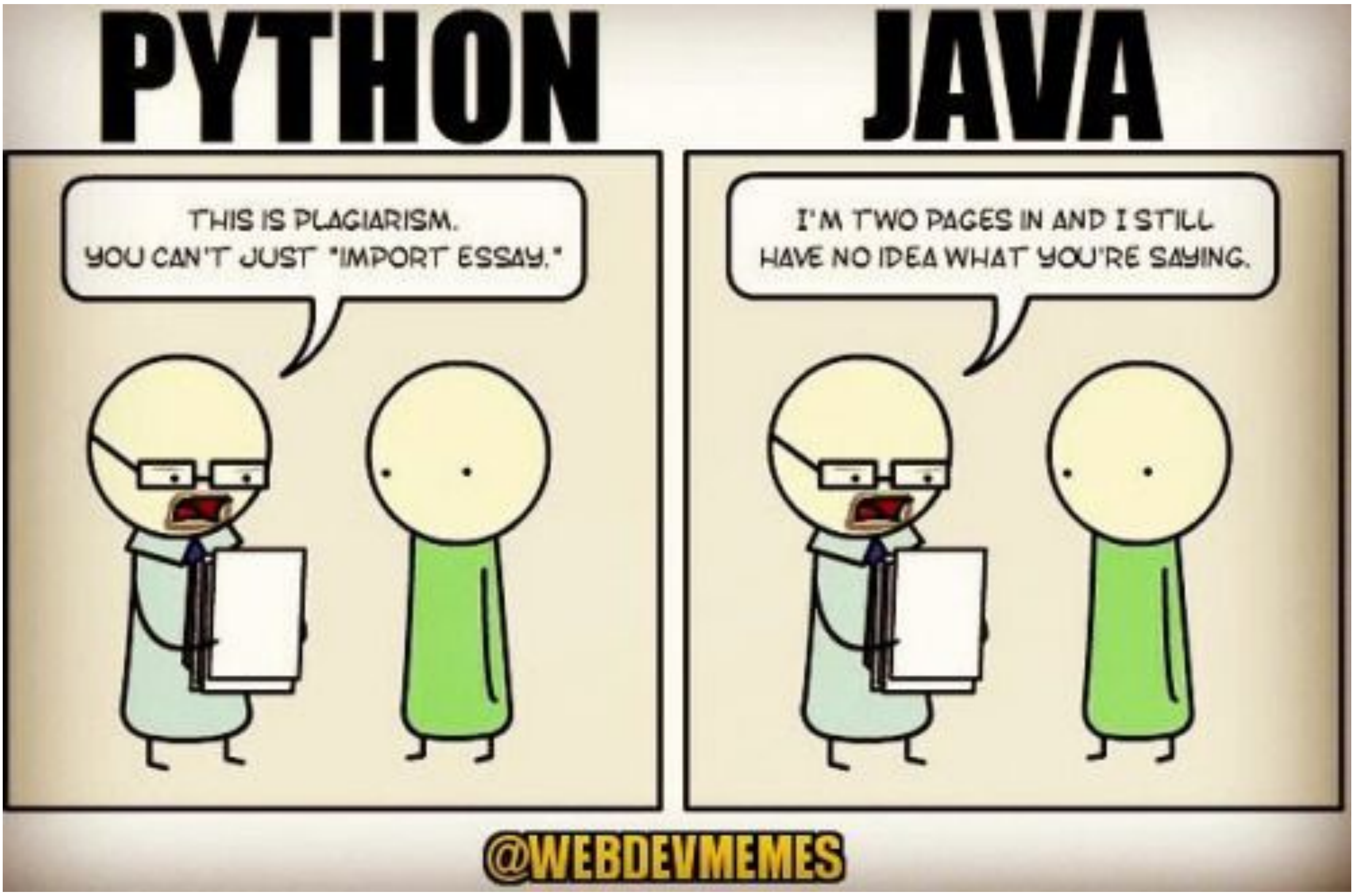
You can use the following *Scanner* methods to read in particular chunks or variable types:

- keyboard.nextLine(): reads the entire line, including the newline character
- keyboard.next(): reads in one word as a *String*
- keyboard.nextInt(): reads in a number and converts it to *int*
- keyboard.nextDouble(): reads in a number and converts it to *double*

The above methods skip over any whitespace (spaces, tabs, and newlines) and read one 'word' at a time. You will notice an error is thrown if the user input does not match the expected type.

See the *Scanner documentation* for more methods.

Give it a go and modify the code snippet above to ask how old your user is!


PYTHON
JAVA
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY."
I'M TWO PAGES IN AND I STILL HAVE NO IDEA WHAT YOU'RE SAYING.
@WEBDEVMEMES

Relevant Reading Resources

Relevant Reading Resources

1. WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 1)
2. SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 1)
3. About the Java Technology (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html.
4. Gosling, J., & McGilton, H. (1995). The Java language environment. Sun Microsystems Computer Company, 2550, 38. Available at: https://www.oracle.com/java/technologies/language-environment.html.