Java SE

Presentation layer, website (What user interacts with), Angular

Logic layer / business layer – hard calculations are made (front end, customer interacts it, logic layer distinguishes what is to be shown from the database.) it is the workhorse.

Persistence Layer/ Database Layer- store any information that we want to process, MySQL, Springboot formulates a response to give to the front end / presentation layer.

-Java is platform depended due to java virtual machine (JVM) may vary depending on platform

-It will work on any machine that can run the JDM

-We will be writing source code (Technical), Compiled Language. The source code is converted to binary / machine code by JVM when it is compiled.

Java is Statically typed. We need to give it a TYPE.

String is a word.

Variables and Methods

Variables

Int, double, float, string

* Variable is a container that holds values that we use in the program.
* Every variable has a data type and a name associated with it, and eventually, a value.
* A variable could be declared to use one of the eight primitive types.

Methods

(How we get things done in java, a set of instructions we can call), how we retain usability

* A method is a set of code which is referred to by name and can be called at any point in a program by simply utilising the methods name.
* Think of a method as a sub-program that acts on a data and often returns a value.
* Every method has a name and a return type, and can also have parameters that are variables that you give it that it may need to perform its function.

Return type void does not return anything.

Naming Conventions

-It is important to be aware of the naming conventions associated with a language when starting out.

Here are the key ones for java:

1. no spaces in names

2. Classes begin with capital letters

3. Methods are attributes begin with lower case

4. Methods and attribute names are in camel case (**CamelCase)**

Syntax

-Curly braces {} think of scope, they denote scope, the describe on a level things that are assigned within that scope.

; used to end a line of code, sometimes it won’t be needed.

// comment a line

/\* \*/ used to surround comment blocks (don’t write comments that we know what they do from reading)

/\*\* \*\*/ used to surround documentation comment blocks

( ) used to surround parameters and arguments

. (dot notation) used to access a variables methods and attributes.

-Make variable names sensible.

Examples

Private (Access modifier) void (Return Type) example method (Method Name) (scope/ method body ){ - method name

}

Int (Variable type) exampleInt (Variable Name) = 3; - end of statement

ExampleInt = 40; - Overriding the variable

Examples – Parameters and Returns

Private (Access Modifier) int (Return Type) exampleMethod(int exampleParameter – putting the parameter that we want this method to take) {

ExampleParameter = exampleParameter + 1;

Return exampleParameter; -returns a variable of the data type that was declared before, if the return statement was not there the code would not work

System.out.println(exampleMethod(400));

//Will output 401

Example – The flow of execution

Main method

Public static void main(String[] args)

{

Method1(); - set of instructions, it will complete them before moving to method 2

Method2();

System.out.println(method 3());

}

Static void method1(){

System.out.println(“hello”);

}

Static void method2(){

System.out.println(“world”);

}

Static method3(){

Return “!”; -this is preferred method to mthods 1 and 2

}

Primitive Data Types

Boolean – true of false, on or off -imp

Byte – 8 bit

Char – single character denoted with single apostrophise ‘B’ -imp

Int – integer number -imp

Double- decimal numbers, 32 bit -imp

Floats are smaller than double, 64 bit

Operators

Assignment, =

Arithmetic , + - / \* %-modulus, it returns the remainder.

Unary

+ indicates a positive value

-Indicates a negative value

++ increments by 1

-- decrements by 1

! the not operator, inverts the value of a Boolean

Scope

-Class Level/Instance Scope

Variables that are referenceable throughout the entire class, these methods are inside the class but outside of methods.

Generally defined at the top of the class.

-Method / local scope

Variables that are temporary and only used in the method they are declared in.

As soon as the method ends all variables declared inside that method are no longer referenced too and cannot be accessed any more.

-Loop Scope

Variables that are declared inside a loop declaration, only accessible inside the loop and are lost..

Arrays

Single dimensional array [1][2][3][4]

used to hold collection of dataifferent types of arrays and collections are intended for different situations.

One way to create an array is to use a set of square brackets:

**int**[] arrayOfInts = {5,6,33,45,50};

You can specify the length by putting the length in square brackets when instantiating the array:

**int**[] arrayOfInts = **new int**[5]; (NEW, WHEN WE ARE CREATING A NEW OBJECT,INSANTIATING).

You can assign values directly to an index by specifying the index in square brackets: arrayOfInts[3] = 45;

Multi-Dimensional Array

These are arrays where each   
index of an array holds another array.

These can be specified by having two sets of square brackets:

**int**[][] multiArray = {{5,6,9},{25,70,5},{8},…};

You can specify the length in the same way as in single dimension arrays:

**int**[][] multiArray = **new int**[3][2];

You can assign values to an index by specifying both indexes:

multiArray[1][2] = 5;

For Each Loop

* For-Each loops iterate through each element in an array or collection.
* This is a much tidier version of what regular for loops can achieve, both have their pros and cons.
* “For every integer **i** in **num**, do **x**”

int num[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

for (int i : num) {

System.*out.println("Number: " + i);*

List Implementations

* The **List** interface extends **Collection.**
* Precise control over where in the list each element is inserted.
* User can access elements by their integer index or via searching for elements in the list.
* Provides a special iterator called **ListIterator**
* Some **List** implementations prohibit null elements, some have restrictions on their type of elements
* **LinkedList** is another commonly used implementation of **List**
* **ArrayList –** Index based system
* **Linked list –** Doubly Linked list system
* Because of these systems they implement they have their own strengths.
* **ArrayList –** Faster searching/Lower overhead
* **Linked list –** Faster deletion/Insertion

IntArray[2]=45; this changes the 3rd cell in the array, [0 , 1 , 2 , 3]

List is an ordered Collection

ListIterator allows for element insertion and replacement, bidirectional access in addition to the normal operations that the Iterator interface provides

The searching functions that List provides are extremely inefficient and should be used with caution.

Don’t assume just because List allows null elements that an implementation of List will as well.

ArrayList maintains an index based system for its elements which implicitly makes it faster for searching for an element in the list.

Linked list implements a doubly linked list system which requires the traversal through all the elements to search for an element.

Linked list stores extra information (positions/neighbour nodes) – High memory consumption

Linked list deletion only requires the change of the neighbour node pointers whilst array list requires all data to be shifted to fill the space.

Allow access to objects that are stored in them

Used to store data

Simple data structure

0 index based system

Talk about how when we delete something from an arraylist everything gets shifted down to fill the new gap has been made – hence the reason that it is slow at adding and removing – because it has to reindex itself eveytime

In a linked list everything effectively is linked to something else

Searching for things in a linked list is extremely slow

The benefit is with if you want to remove something

All it effectively has to do is change to references

* The thing that pointed to it
* And the other thing

Remember auto garbage collection

Remember the diagrams with the grid of circles for array list

And the circles linking to one another for linked list

The arrays shown previously have been primitive arrays.

Seeing as Java is an Object orientated language you will most likely need arrays that can store Objects.

The most common array you will use will be the ArrayList:

ArrayList<Object> objects = **new** ArrayList<>();

There are many other types of array such as the Map, Set & List which are each tailored to different scenarios.

An ArrayList is a dynamic data structure, this means items can be added and removed at will, changing the size of your arraylist as opposed to normal arrays where you are stuck with a static initial size of your array.

f you know how many items you want to store, you use arrays unless you really need some specific functionality (which you could debatably create yourself) that arraylists offers or speed/memory efficiency is of major concern.

If not, use arraylists.

Lists allow duplicate values

Some of the commands are slightly different, .length is now .size

.get to get an element

.set to change an element (this will take 2 parameters)

Cant store int/primitive numbers in a list , wrapper class <Integer>

Maven Project

Group Id: com.qa

Artifact Id: -same as project

Mvnrepository.com has all of the dependencies in it. – external library, provides us with functionality without having to do it.

Streams

* They may sound similar to input and output streams but they are **different**.
* A Stream is an abstraction, it is not a place where we can store information.
* It does not hold data, but it can manipulate it.
* Streams have effectively replaced for each loops in Java 8.

A stream is an abstract concept

Collection-raise it to a stream. At each point in the stream we can add different things for example, filter, map,, reduce, collect

* A Stream represents a **pipeline** through which the data will flow and also the functions that will operate on the data.
* We can ‘raise’ a collection to a stream by calling the .stream() method on a collection.
* Once we have done this we can call functions that will interact with the stream.
* This will look like multiple methods being chained together.

Some Fundamental Functions

* [**Filter**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) returns a new stream that contains **some** of the elements of the original.
* It computes which elements should be included, think checking a condition.
* In the functional style we don’t bother with *ifs*, we filter the stream and work only on the values we require.
* [**Map**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) transforms the stream elements into something else, it accepts a function to apply to each and every element of the stream and returns a stream of the values the parameter function produced.
* This is the bread and butter of the streams API, map allows you to perform a computation on the data inside a stream.
* **FlatMap** can work with other collections, for example if I am working through a stream of String[ ] then I will need to use flatmap to get the information from each of the String [ ].
* **Peek** is mainly used for debugging purposes.
* It enables us to look at what is happening in the stream as it flows from one point to another
* [**Reduce**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) (also sometimes called a fold) performs a reduction of the stream to a single element. You want to sum all the integer values in the stream – you want to use the reduce function.
* You want to find the maximum in the stream – reduce is your friend.
* [**Collect**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) is the way to get out of the streams world and obtain a concrete collection of values.
* For example a List.
* List<String> **strings = Arrays.*asList("abc", "", "bc", "efg", "abcd", "", "jkl");***
* List<String> **filtered = strings.stream().filter(string -> !string.isEmpty()).collect(Collectors.*toList());***
* The List filtered, now only contains (“abc”,”bc”,”efg”,”abcd”,”jkl”).
* List<Integer> **numbers = Arrays.*asList(3, 2, 2, 3, 7, 3, 5);***
* numbers.stream().sorted().forEach(**System.*out::println);***
* We do not need to use all of these functions every time we work with streams.
* We create our own functionality to interact with the stream, called Lambdas.

Lambdas

* Java lambda expressions are Java's first step into functional programming.
* A Java lambda expression is thus a function which can be created without belonging to any class.
* A Java lambda expression can be passed around as if it was an object and executed on demand.
* In Java, lambda expression is **“SAM type”**, which is an interface with a single abstract method (interfaces can now include non-abstract methods – the ***defender*** methods).
* There are some existing interfaces we can use.
* Or we can write our own interfaces to support lambda expressions.
* These are functional Interfaces, an interface with **only 1** abstract method.

Good website to go over lambdas:

<http://www.java2s.com/Tutorials/Java/Java_Lambda/index.htm>