Introduction to Java

Why the name Java?

Java was initially called Oak after an oak tree that stood outside James Gosling's office; it went by the name Green later, and was later renamed Java, from Java coffee, said to be consumed in large quantities by the language's creators. This also tells the story behind the logo of java. [https://www.quora.com/Why-is-Java-called-so]

First Steps

1. Install Java JDK - *It's the full featured Software Development Kit for Java, including JRE, and the compilers and tools (like JavaDoc, and Java Debugger) to create and compile program.*
2. Visit https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html to read about installation on your platform. Smasll examples may also be executed online on https://www.ideone.com
3. Install Eclipse (Check how many of them have experience with eclipse) from: https://www.eclipse.org/downloads/eclipse-packages/

Eclipse is not necessary! It is a good IDE for JAVA. Other options are Netbeans https://netbeans.org/downloads/ or IntelliJIDEA https://www.jetbrains.com/idea/

Hello World

1. Both javac and java must be available in the system.

```
java -version
```

should shows something like the following:

```
java version "1.8.0_05"
Java(TM) SE Runtime Environment (build 1.8.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.5-b02, mixed mode)
```

HelloWorld.java

```
public class HelloWorldApp
{

  public static char bar(int a, int b){
   return (char)(a+b);
  }
  // arguments are passed using the text field below this editor
  public static void main(String[] args)
```

```
  {
    String s="Hello Worl";

    System.out.println(s+bar(50,50));

  }
}
```

- Note that Java is *case senitive*
- Compile using the `java` compiler.
- Note that compiled `java` code is not native code, infact techincally speaking that is not a compiler. Jacav compiler translates java code into something called *bytecode* which the underlying java VM can understand and executes. The bytecode then is compiled to machine-code using a Just-In-Time compiler within the JVM. The Java Virtual Machine is a sort of a minicomputer (that's why it is called a virtual machine) that executes your Java bytecode.

Hello World 2

Importance of commenting the code Especially when working in a team.
Documentation will be evaluated.

Use `java.util.Scanner` to read input from stdin.

```
import java.io.Console;
import java.util.Scanner;

public class HelloWorld {

    //input reader
    final Scanner s = new Scanner(System.in);

    //read from stdin
    public void fancyCalculator(){
        System.out.println("JAVA first calulator");
        while(true){
            //operands
            int l,r;
            System.out.println("Type the 1 operand");
            l=s.nextInt();
            System.out.println("Type the 2 operand");
            r=s.nextInt();
            //output the sum of the two number
            final int ans = l+r;
            System.out.print("-> ");
            System.out.println(ans);
        }

    }
```

```
    public static void main(String[] args) {
        HelloWorld calculator = new HelloWorld();
        calculator.fancyCalculator();
    }
}
```

Hello Classes

A class is a descrition both the **state** and the **behaviour** of an Object. Think about the state of a traffic light. At any point what are the possible state a traffic light can be?

Do computer programs have a state? If yes what are the *variables* of a computer program state?

Bookstore

What are getters and setters? Show how to automatically generate them using *eclipse*

- *Source -> Generate Getters and Setters*

```
package BookStore;

import java.util.Date;

public class Book {

    private String author;
    private Date year;
    private int npages;
    private String ISBN;


}


package BookStore;

public class BookStore {

    private Book[] library;

    /*number of book inserted*/
    private int size;
    /* Maximum capacity of the bookstore*/
    private int capacity;

    public BookStore(int _capacity) {
        capacity = _capacity;
        assert(capacity>0);
```

```
        library= new Book[capacity];
        size=0;
    }

    public void addBook(Book b){
        library[size] = b;
        size++;

        //short version
        //library[size++] = b;
    }

    public void removeBook(Book b){

    }

    public void removeBook(int b){

    }

    public void searchForBook(Book b){

    }
}
```

How to implement the remove method?

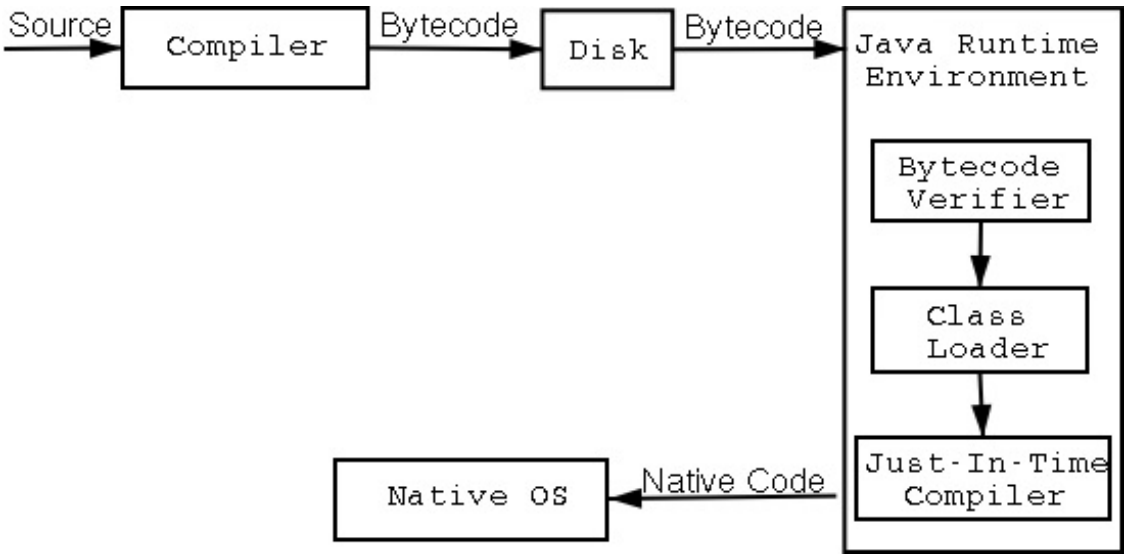
What about the search method?

Java Basics

Java is **Object Oriented** ans so supports concepts like:

- Classes
- Objects
- Polymorphism
- Inheritance
- Encapsulation
- Abstraction

Inline-style:



- Compile: `javac HelloWorld.java` producing the bytecode `HelloWorldApp.class`
- Execute using the `java JRE` (Java Runtime Environment): `java HelloWorldApp`

Bytecode - What's under the hood?

- See what `javac` has generated using an hex editor of your choice (`bless` or `xxd` on Linux might be a good choice)

```
CA FE BA BE 00 00 00 34 00 1D 0A 00 06 00 0F 09 00 10 00 11 08
00 12 0A 00 13 00 14 07 00 15 07 00 16 01 00 06 3C 69 6E 69 74
3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65
4E 75 6D 62 65 72 54 61 62 6C 65 01 00 04 6D 61 69 6E 01 00 16
28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 29
56 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 01 00 12 48 65 6C 6C
6F 57 6F 72 6C 64 41 70 70 2E 6A 61 76 61 0C 00 07 00 08 07 00
17 0C 00 18 00 19 01 00 0C 48 65 6C 6C 6F 20 57 6F 72 6C 64 21
07 00 1A 0C 00 1B 00 1C 01 00 0D 48 65 6C 6C 6F 57 6F 72 6C 64
41 70 70 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63
74 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01
00 03 6F 75 74 01 00 15 4C 6A 61 76 61 2F 69 6F 2F 50 72 69 6E
```

- Each pair of number (1 byte) is directly translatable to an instruction. For instance the highlighted `3C` translates to *store int into variable 1*. Read here for more informations: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
- `javap -c` translates bytecode to a mnemonic representation saving us an headache. Try `javap -c HelloWorldApp.class`

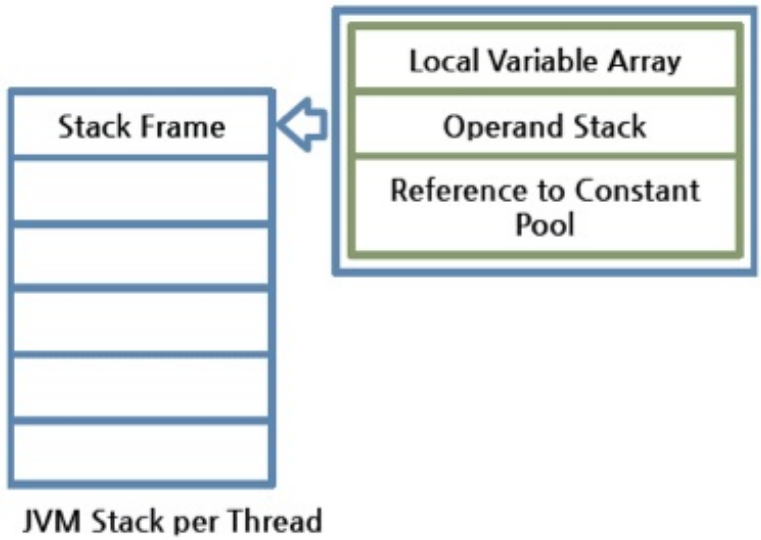
```
Compiled from "HelloWorldApp.java"
public class HelloWorldApp {
  public HelloWorldApp();
    Code:
       0: aload_0
```

```
      1: invokespecial #1                  // Method java/lang/Object."<
      4: return

  public static char bar(int, int);
    Code:
      0: iload_0
      1: iload_1
      2: iadd
      3: i2c
      4: ireturn

  public static void main(java.lang.String[]);
    Code:
      0: ldc           #2                  // String Hello Worl
      2: astore_1
      3: getstatic     #3                  // Field java/lang/System.out
      6: new           #4                  // class java/lang/StringBuil
      9: dup
     10: invokespecial #5                  // Method java/lang/StringBui
     13: aload_1
     14: invokevirtual #6                  // Method java/lang/StringBui
     17: bipush        50
     19: bipush        50
     21: invokestatic  #7                  // Method bar:(II)C
     24: invokevirtual #8                  // Method java/lang/StringBui
     27: invokevirtual #9                  // Method java/lang/StringBui
     30: invokevirtual #10                 // Method java/io/PrintStream
     33: return
}
```

- Take a closer look to the `bar` function.
- Note the # as operand of some functions. Those are references to *constant objects in the constant pool of the class*.
- Run `javap -c -s -verbose HelloWorldApp.class` to take a closer look to that.
- Number at left of the instructions represent their position within the **frame**. Each thread in the JAVA execution mode (and of most of the today's languages) has a so called **stack** which stores **frames**. A frame is created whenever a method (that's name for functions in java) is created. It consists of three main parts: an array of local variables, a set of references to a pool of constant objects and a stack of operands. The stack is used to push or pop values within the lifetime and execution of the method.For instance the bar method loads two elements on the stack (presumibly the operands) and then invokes the `iadd` function which actually performs the addition. -The following is the actual bytecode for the bar method: `1A 1B 60 92 AC`



Examples

Example - 1 - Basic Sorting

Allocate a vector of int of size N (read from stdin). Populate it using Random positive numbers in the interval [1..N]. Write two method for sorting arrays. - Bubble sort (5 mins) - Merge Sort (15 mins)

Sort the array using the aforementioned methods.

Example - 2 - Sorting using Collections

Insted of using a basic array, use a `Vector` and run the java Collestion's built-in sorting method to sort the array `Collections.sort`.

```
static
<T extends Comparable<? super T>>
void
sort(List<T> list)
          Sorts the specified list into ascending order, according to the
static
<T> void
sort(List<T> list, Comparator<? super T> c)
          Sorts the specified list according to the order induced by the
```

- Note that you can use your own comparator.
- Write a custom comparator and sort the Collection using it.
- Under the hood, `Collection.sort` uses merge-sort which is known for having complexity of `O(nlogn)`. **What is the requirement on the complexity of the comparator in order for the overall complexity of the sorting procedure not to grow?**
- *What happen if you want to sort a custom class which requires* ***linear-time***

comparison?

Example - 2 - BookStore

...

Creating and Destroying Objects

Static Factory Methods

- Allows construction of instances of class with methods that have a **meaningful** name
- They are not required to create a new object each time they are invoked. They can also manage cache of instance of a class to improve performance. Image how much this can be helpful in case construction is expensive.
- They can return object of any subclass or interface. `EnumSet` implementation for instance has no public constructor but only a static factory method which rturn two different `EnumSet` implementation according to the size of the enum. The client only agree that the returned object impelements a common interface. If the developer adds one or more case client code dœs not change!
- Class that do not provide public or protected constructor cannot be subclassed!
- Constructors stand out in documentation while static factory methods are just normal methods which sometimes can be hard to spot, especially in large and complex class documentation. There are convention about the name of factory methods that might help on this. Common names are
- `valueOf` act like a type conversion type (See Boolean)

-getInstance which return an instance described nby the parameters. Singleton uses this naming heavily.

- `newInstance` which garantee (in a opposite fashing with singleton) that all objects are different from the others

Take with you: factory methods are useful so consider them before providing public constructors!

Builder

Static factory methods and constructors share the same limitation. They do not scale when the number of paramtersw getr larger and larger. When you have large number of parameters what you usually do is provide constructors with *one*, *two*, *three*, etc. number of parameters so the client can invoke the less wordy one. This approach is not optimal though in case objects with many parameters (100 for instance) and are very hard to read. Alternatively you can use the Java Bean Approach where constructors are not provided and client manually set fields. This approach works but requires client to write a lot of code (possibly). Plus, since construction is not atomic (client makes a number of calls to `set` methods) the object might be in a inconsistent state during

construction. Checking the validity of the parameters is harder since the check has to be decoupled from the construction. Beans cannot be **immutable**, by definition.

- Builder mix the advantages of both approaches, safety of *telescoping* and readibility of Beans.

```
// Builder Pattern
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }
        public Builder calories(int val) {
            calories = val;
            return this;
        }
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public Builder carbohydrate(int val) {
            carbohydrate = val;
            return this;
        }
        public Builder sodium(int val) {
            sodium = val;
            return this;
        }
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }//builder

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
```

```
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }

}
```

Using `Builder` is easy since you create a builder for a class and using Beans stile calls you set only the parameters needed and then the `build()` calls take care of checking invariants, consistency of required and optional parameters. If any invariants is violated then it might raise an exception. Builder is flexible. This about how it can fills some fields automatically, possibly based on some input parameter (for instance the first `N` parameters using subsequents number`).