Intro

Friday, May 3, 2024 8:05 AM

Java is Not Javascript

- **Java** is a compiled/interpreted language
- Backend

- **Javascript** is an interpreted language
- Mostly used on web browsers and some backend with NodeJS

Compiler

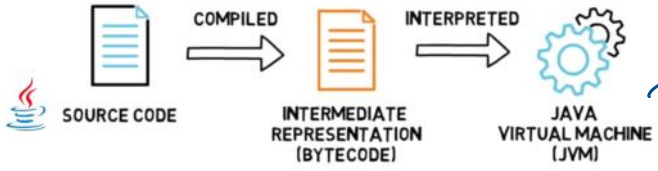
- A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

Interpreter

line by line execution

- The interpreter reads each statement of code and then converts or executes it directly

Compiled	Interpreted
Code can be executed directed by computer's cpu	Another program executes code
Code must be transformed into machine readable instructions	No transformation needed
Runs faster	Slower
Better Performance	Faster for Development



The compiled bytecode is executed by JVM

Compiled languages

Compiled languages are programming languages that are typically converted into machine code before they are run. This machine code can be directly run on the target platform or it can be used to produce an executable file.Compiled languages tend to be faster than interpreted languages because the conversion to machine code happens just once. This means that there is no need to parse and interpret the code every time it is run, as is the case with interpreted languages. In addition, compiled languages often have stricter rules about syntax and semantics, which can result in more reliable and efficient code. However, compiled languages can be more difficult to work with than interpreted languages because errors are usually only detected at compile time, not at runtime. This means that it can be harder to debug compiled code.

Some languages, such as Java, can be either compiled or interpreted, depending on how they are configured.

Interpreted languages

There are two types of programming languages: compiled and interpreted. Compiled languages are translated into machine code, which is then run on a computer. Interpreted languages, on the other hand, are not directly translated into machine code. Instead, they are run through an interpreter, which translates the code into instructions that can be executed by the computer.

One advantage of interpreted languages is that they are usually easier to learn and use than compiled languages. This is because interpreted languages tend to be more high-level, meaning that they are closer to human language and require less detailed instructions than compiled languages. As a result, interpreted languages are often more user-friendly and easier to read and write.

Another advantage of interpreted languages is that they tend to be more portable than compiled languages. This is because an interpreter can be written for multiple platforms, meaning that the same code can run on different types of computers. In contrast, a program written in a compiled language can only run on the type of computer for which it was compiled.

There are some disadvantages to using interpreted languages as well. One drawback is that they tend to be slower than compiled languages because the code must be passed through the interpreter each time it is run. Another disadvantage is that interpreters can be harder to debug than compiled programs because it can be difficult to track down errors in the interpretation process. Overall, though, interpreted languages have some clear advantages that make them worth considering for many programming tasks.

Static vs Dynamic Type Checking

There are two main approaches to type checking in programming languages: static and dynamic. Static type checking is when types are checked during compilation, before the program is run. This means that any type errors will be caught early on and can be fixed before the program is deployed. Dynamic type checking, on the other hand, happens at runtime. This means that type errors can go undetected until the program is actually being used.

Which approach is better depends on various factors. In general, static type checking can catch more errors and can help to prevent bugs from getting into production code. However, it can also add more of a burden on developers, as they have to make sure that their code is correctly typed before it can be compiled. Dynamic type checking, on the other hand, is usually more flexible and easier to use, but it can lead to more runtime errors.

statically typed → Java, Typescript
dynamically typed → Javascript, Python

eg:- Java

```java
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
        int brand = "Amigosocode";
        System.out.println(brand);
    }
}
```

code won't even run

JS

```
1 for (let i = 0; i < 10; i++) {
2     console.log(i)
3 }
4 foo
5 var brand = "Amigoscode";
6 console.log(brand);
```

```
6
7
8
9
/Users/amigoscode/code/imajid/src/foo
foo
```

Code runs but gives Runtime error

Type	Size (bits)	Minimum	Maximum	Example
byte	8	-128	127	byte b = 100;
short	16	-32,768	32,767	short s = 30_000;
int	32	-2147483648	2147483647	int i = 100_000_000;
long	64	-9223372036854775808	9223372036854775807	long l = 100_000_000_000_000;
float	32	-2-149	(2-2-23)·2127	float f = 1.456f;
double	64	-2-1074	(2-2-52)·21023	double f = 1.456789012345678;
char	16	0	216− 1	char c = 'c';
boolean	1			boolean b = true;

Pass by Value & Pass by Reference

primitives ⇒ pass by val

objects ⇒ pass by ref

In Java, there are two ways to pass arguments to methods: **pass by value** and **pass by reference**.

When you pass an argument by value, the method makes a copy of the argument and uses that copy as the actual parameter. This means that any changes made to the parameter within the method will not be reflected in the original argument.

On the other hand, when you pass an argument **by reference**, the method accesses the original argument directly. This means that any changes made to the parameter within the method will be reflected in the original argument.

eg:-

Copy is passed for primitives

```java
static int adder(int num) {
    return ++num;
}

public static void main(String[] args) {
    int num1 = 1;
    int result = adder(num1);  // num1 is passed as a copy - by val

    System.out.println(STR."num1 is \{num1} and the result is \{result}");

}
```

```
"C:\Users\Chamika Jayasinghe\.jdks\openjdk-22\bin\java.exe" --enable-preview "-javaagen
num1 is 1 and the result is 2
```

Reference to the obj. in heap is passed

```java
static class Cat {
    //      attributes
    4 usages
    private String name;

    //      constructor
    1 usage
    public Cat(String name) {
        this.name = name;
    }

    // Getter for name
    no usages
    public String getName() {
        return name;
    }

    // Setter for name
    1 usage
    public void setName(String name) {
        this.name = name;
    }

    2 usages
    public void meows() {
        System.out.println(STR."\{name} says: Meow!");
    }
}
```

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

    Cat c1 = new Cat( name: "Susy");
    Cat c2 = c1;  // pass by reference ie a reference to the object is passed here
    // Both c1 and c2 reference the same object

    c1.meows();  // Susy says: Meow!
    c2.setName("Susanna");
    c1.meows();  // Susanna says: Meow!

}
```

Access Modifiers

There are four access modifiers in Java:

- public
- private
- protected
- default.

The public modifier is the most accessible, while the private modifier is the least accessible.

The default modifier is only accessible within the same package.

The protected modifier is accessible within the same package and all subclasses. In general, it is good practice to make fields and methods as accessible as possible, so that other classes can make use of them. However, there are some cases where it is necessary to restrict access in order to protect data from being modified or accessed incorrectly.

Access modifiers can help to ensure that data is only accessed and modified in the way that is intended.

Modifier	Class	Package	Subclass	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
no modifier / default	✅	✅	❌	❌
private	✅	❌	❌	❌

Static keyword :- static attributes & methods belong to the class itself, not to an instance.

```java
public class Main {

    2 usages
    public static String var1 = "John";  // static variable
    2 usages
    public String var2 = "Smith";  // * instance variable

    public static void main(String[] args) {  // * static method , so inside a static method, we can only access static variables and methods directly,
     without creating an instance of the class

        System.out.println(var1);  // since var1 is static, it can be accessed directly

        Main m = new Main();

        System.out.println(m.var2);  // var2 is not static, so it can only be accessed via an instance of the class

        m.myMethod();  // myMethod is not static, so it can only be accessed via an instance of the class

    }

    1 usage
    void myMethod() {
    }  // * instance method

}
```

creating an instance so that non-static methods/attributes can be accessed within the static context.

```java
package com.chamika.mypackage;

import com.chamika.Main;

1 usage
public class Mine {

    1 usage
    Main m1 = new Main();

    no usages
    String name = Main.var1;  // var1 was static , so it belongs to class

    no usages
    String x = m1.var2;   // var2 was not static, so it belongs to an instance rather than the class itself


}
```

final keyword

The final keyword in Java is used in different contexts.

It can be used in the context of a variable, a method or a class.

- When the final keyword is used with a variable, you can no longer change the value of that variable.
- When the final keyword is applied to a method, you will no longer be able to override that method.
- When the final keyword is applied to a class, you will no longer be able to extend that class.

1. **Final Variables**: When you declare a variable as `final`, you're stating that its value cannot be changed once initialized. This is useful when you want to create constants or ensure immutability.

java — Copy code

```java
public class FinalExample {
    final int MY_CONSTANT = 10;

    public void modifyVariable() {
        // This will result in a compilation error
        MY_CONSTANT = 20;
    }
}
```

2. **Final Methods**: When you declare a method as `final`, you're preventing subclasses from overriding that method. This is commonly used to enforce certain behavior in a class's methods.

java — Copy code

```java
public class Parent {
    public final void finalMethod() {
        System.out.println("This method cannot be overridden.");
    }
}

public class Child extends Parent {
    // This will result in a compilation error
    public void finalMethod() {
        System.out.println("This won't compile!");
    }
}
```

3. **Final Classes**: When you declare a class as `final`, you're indicating that it cannot be subclassed. This is often used for classes that should not have any subclasses, such as utility classes or classes that are already complete and should not be extended.

java — Copy code

```java
final public class FinalClass {
    // Class members and methods
}

// This will result in a compilation error
public class SubClass extends FinalClass {
    // Trying to extend a final class is not allowed
}
```

Enum

Enums, short for enumerations, in Java provide a way to define a set of named constants. They are used when you have a fixed set of values that a variable can take. Enums are more type-safe than constants or integers because they restrict a variable to having one of a predefined set of values.

```
public enum Day {
    1 usage
    SUNDAY( abbreviation: "Sun"),
    no usages
    MONDAY( abbreviation: "Mon"),
    no usages
    TUESDAY( abbreviation: "Tue"),
    no usages
    WEDNESDAY( abbreviation: "Wed"),
    no usages
    THURSDAY( abbreviation: "Thu"),
    no usages
    FRIDAY( abbreviation: "Fri"),
    no usages
    SATURDAY( abbreviation: "Sat");

    2 usages
    private final String abbreviation;

    // *constructor
    14 usages
    Day(String abbreviation) {
        this.abbreviation = abbreviation;
    }

    1 usage
    public String getAbbreviation() {
        return abbreviation;
    }

}
```

```
public class Main {


    public static void main(String[] args) {

        Day today = Day.SUNDAY;

        System.out.println(STR."Today is \{today.getAbbreviation()}");


    }

}
```

Type casting

1) Implicit type (widening) casting
2) Explicit type [Narrowing] casting

```
//      * Implicit casting - widening
int myInt = 20;
double myDouble = myInt;
System.out.println(myDouble);

//      * Explicit casting - narrowing
double x = 34.45;
int y = (int) x;
System.out.println(y);
```

```
Main ×
"C:\Users\Chamika Jayasinghe\.jdks\openjdk-22\bin\java
20.0
34
```

• Wrapper Classes

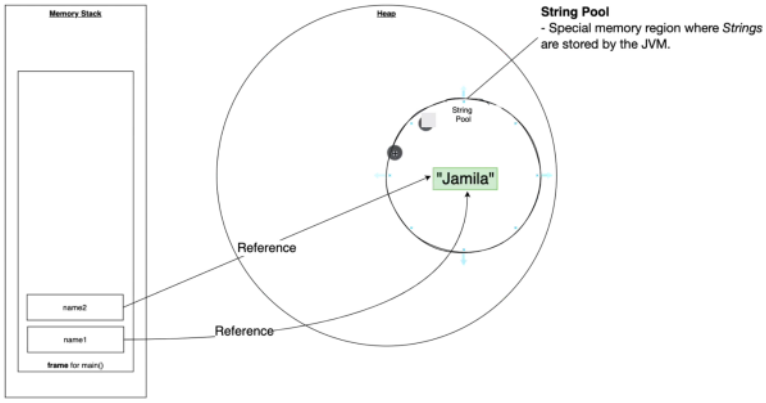
- Wrapper classes allow primitive data types to be treated as objects.
- They provide methods to convert primitive types into objects and vice versa.
- They are useful when working with collections, generics, and methods that require objects.
- Wrapper classes also provide utility methods and constants for each primitive type.

```
Integer i1 = 20;
Double d1 = 34.56;

String age = "21";
int age_int = Integer.parseInt(age);
System.out.println(STR."I am \{age_int} years old");

int max_num = Integer.max(i1, age_int);
System.out.println(max_num);
```

```
I am 21 years old
21
```

String

```java
//       *  String pool will be checked, since Jamila exists already , both name1 and name2 in String pool will point to same object in the pool inside the heap
      String name1 = "Jamila";
      String name2 = "Jamila";
```

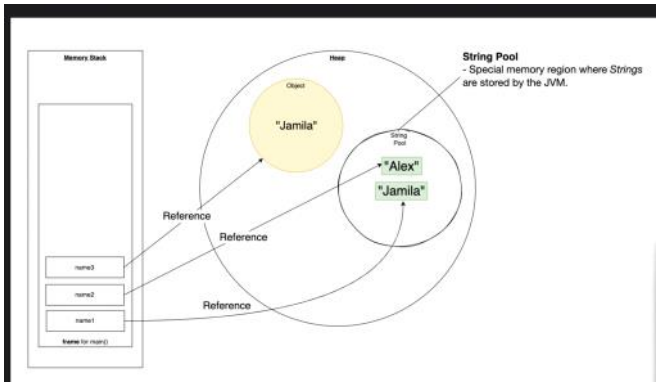


String Pool

When a string is created, the JVM (Java Virtual Machine) allocates memory for that string in the heap. This memory is used to store the string object and its associated character data.

The `string pool` is the area of heap memory where strings are stored. When a string literal (i.e. a string enclosed in double quotes - `String foo = "foo";`) is created, the JVM first checks if there is already an identical string in the pool. If so, the reference to that string is returned. If not, a new string object is created and added to the pool. As a result, strings created using string literals tend to be more efficient in terms of memory usage than strings created using the String class constructor. However, it should be noted that strings created with the String class constructor (`String foo = new String("foo");`) are not added to the `string pool`. Therefore, if you need to create strings that are not shared between multiple objects, you should use the String class constructor rather than string literals.

```java
//       *  String pool will be checked, since Jamila exists already , both name1 and name2 in String pool will point to same object in the pool inside the heap (These strings will live inside the String Pool since they are string literals)

      String name1 = "Jamila";
      String name2 = "Alex";

//       * This string will live inside the heap, but not inside the pool
      String name3 = new String( original: "Jamila");
```



String are Immutable

Strings in java are immutable because they are constant. The value of a string cannot be changed once it is created. strings are also thread-safe, meaning that they can be safely used in concurrent programming without the risk of data corruption.

Immutability also makes strings more secure, because they cannot be modified by malicious code.

Finally, strings are more efficient when they are immutable, because the java virtual machine can optimize them better.

String Equality

In Java, strings are compared using the equals() method. This method accepts a String object and returns true if the strings are equal, false otherwise. If you want to compare strings without regard to case, you can use the equalsIgnoreCase() method instead.

It's important to note that strings should never be compared using the == operator. This is because == compares references, not values. In other words, it will only return true if both strings are pointing to the same memory location. This is almost never what you want when comparing strings.

```java
package com.amigoscode;

public class Main {
    public static void main(String[] args) {
        // Strings
        String name1 = "Jamila";
        String name2 = "Jamila";
        name2 = "Alex";
        String name3 = new String("Jamila");

        System.out.println("String equality with ==");
        System.out.println(name1 == name2); // true
        System.out.println(name1 == name3); // false. because name 3 is in a different me

        System.out.println("String equality with .equals");
        System.out.println(name1.equals(name2)); // true
        System.out.println(name1.equals(name3)); // true
    }
}
```

Scanner :)

```java
Scanner scanner = new Scanner(System.in);

System.out.print("Your name:- ");

String name = scanner.nextLine();

System.out.print("Your age:- ");
int age = scanner.nextInt();

System.out.println(STR."\{name} is \{age} years old");
```

Exceptions

creating a custom exception

```java
1 usage
public static class MyException extends Exception {

    1 usage
    public MyException(String message) {
        super(message);  // call the constructor of the parent class
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        try {

//          System.out.println(Integer.parseInt("hello world"));
//          System.out.println(234 / 0);

            throw new MyException("This is from my custom exception");  // intentionally throw an exception

        } catch (NumberFormatException | ArithmeticException e) {
            System.out.println(e.getMessage());

        } catch (Exception e) {  // * catch all exceptions
            System.out.println(STR."An unknown exception occurred! ==> \{e.getMessage()}");
        } finally {
```

creating custom exception

```java
1 usage
public static class MyException extends Exception {

    1 usage
    public MyException(String message) {
        super(message);  // call the constructor of the parent class
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

        try {

//            System.out.println(Integer.parseInt("hello world"));
//            System.out.println(234 / 0);

            throw new MyException("This is from my custom exception");  // intentionally throw an exception

        } catch (NumberFormatException | ArithmeticException e) {
            System.out.println(e.getMessage());

        } catch (Exception e) {  // * catch all exceptions
            System.out.println(STR."An unknown exception occurred! ==> \{e.getMessage()}");
        } finally {
            System.out.println("Bye.....");
        }
```