

# Java Notes

---

Please note that these notes are just collection of my lecture notes and researchs over the internet. They come with **ABSOLUTELY NO WARRANTY**.

---

- **instanceof** is an **operator** which is used to **test** whether given **object** is an **instance of the class** (Or subclass or interface).
- **instanceof** returns a **Boolean** value.

```
...  
...  
if (john instanceof Student) {  
    System.out.println("John is a student");  
}  
...
```

- **strictfp** ensures that you get the **same result** on **floating-point** arithmetics.
- **Synchronized** keyword makes a class or method **thread-safe**. It means that at a certain time, only one thread can access to class(or method). This is called **locking**. Other threads must wait until this locking removed.

```
// From java.util.Vector class  
  
public synchronized void setSize(int newSize) {  
    ...  
}
```

- An interface with only one abstract method is called a **functional interface**. For example, **java.lang.Runnable** is a functional interface and it has only one abstract method **void run()**.

```
@FunctionalInterface  
  
public interface Runnable {  
    public abstract void run();  
}
```

- A **lambda expression** implements a *functional interface*.

- Lambda expressions came with **Java 8**.
- **Lambda expressions** are functions that **do not have to be an instance of a class**.
- Lambda expressions **can be used as parameters**. They behave as objects.
- A basic **lambda expression** looks as:
  - (parameter list) -> (function body)

---

# Classes

---

- A **method declaration** is basically:

- *Modifier*
- *Return type*
- *Method name*
- *Parameter list*
- *Exception list*
- *Method Body*

```
public void aFunction(int param) throws NullPointerException {
    ...
}
```

- **Method Modifier** Types:

- *Public*
- *Private*
- *Protected*
- *Default*

- **Public**: Can be access from **anywhere** in JVM.
- **Protected**: Can be access from the **same class** it's declared **or** from its **child classes**.
- **Private**: Can be access from **only inside** of the class it is declared.
- **Default**: Can be access from the **same package**.
- **Method signature** consist of **method name and parameter list**.
- **Parameter count, type and sequence** is **important**.
- **Return type and exception list** are **not important** for method signature.
- **Methods** are implemented over **stack**.
- In every **method call**, a **frame is created** on stack.
- Java transfers **parameters** to this frame and creates **local variables**.
- When a **call ends, JVM deletes** the frame.
- Java **does not support multi-value return**. If a method has to return multiple value, it may return a **collection**. If values have **different types**, they may be **encapsulated** in a class and an object of that class can be return value.
- Valid **main method** overloadings:
  - `public static void main(String[] args) { }`
  - `static public void main(String[] args) { }`
  - `public static void main(String []args) { }`
  - `public static void main(String args[]) { }`
  - `public static void main(String...args) { }`
  - `public static void main(final String[] args) { }`
  - `public final static void main(String[] args) { }`
  - `public synchronized static void main(String[] args) { }`
  - `public strictfp static void main(String[] args) { }`
  - `final static synchronized strictfp static main(String[] args) { }`
- A **class can extend** the **class** which **contains** the **main method**. (**Inheritance of main method**)

```

class Container {

    public static void main(String[] args) {

        ...

    }

}

class Child extends Container {

    ...

}

```

- Java **does not support user defined operator overloading**. But in background, `+` operator is **overloaded** for **string concatenation**.

```

System.out.println("Hello" + " World");

```

- **Overloading: Same name, different signature.**

```

public int addInt(int param1, int param2) {

    return param1 + param2;

}

public int addInt(int param1, int param2, int param3) {

    return param1 + param2 + param3;

}

```

- **Overriding: Same name and same signature. Different implementation in different classes.**

```

@Override

public String toString() {

    ...

}

```

- **Overloading** is an example for **compile time polymorphism**.
  - **Overriding** is an example for **run time polymorphism**.
  - **Private** methods are **implicitly final** because **no class can access and override** them.
  - **Adding final** specifier to **private** methods may **create conflicts**.
  - **Primitive data types** are just like in **C language**.
  - Every **other data type** is an **Object**.
  - **Objects** are always **references** to a certain **memory location**.
  - Java creates a **new copy** of the **reference** for **parameters**.
-

# Constructors

---

- A **constructor** can **not** be **final**, **abstract**, **static** or **synchronized**.
- If you do **not write** a **constructor**, **compiler** will **create** a constructor **automatically**.
- If a **constructor** has **parameters**, it is called **parameterized constructor**.
- **Constructor** definitions **doesn't have return statements**, but you **may write**.
- **Constructor** **returns an instance to class**.
- **Constructor** does **not return void**.
- **Constructor name** and **class name** must be **same**.

```
public class Person {  
  
    public Person() {  
  
        ...  
  
    }  
  
}
```

- **Constructors** can be **overloaded**.

```
public class Person {  
  
    String name;  
  
    public Person() {  
  
    }  
  
    public Person(String name) {  
  
        this.name = name;  
  
    }  
  
}
```

- Different from other methods, **constructors** are **invoked** during **only object creation** with **new** keyword. **Other methods** can be **called multiple** times.
- If you **add return type** to **front** of a **constructor**, it **behaves** as any other **method**. But **compiler** will give you a **warning: Method has constructor name**
- You may **create private constructors**.
- **Private constructors** can be used for **singleton class** or **internal constructor chaining**.
- **Constructor chaining**: Calling **super** constructor or **this** constructor.

```

public class Person {

    int age;

    String name;

    private Person(int age) {

        this.age = age;

    }

    public Person(String name, int age) {

        this(age);

        this.name = name;

    }

}

```

- **No-Args constructor != Default Constructor**
- Every class **needs** a **constructor** but you do **not** have to write **destructor**. Because Java **has garbage collection**.
- **Singleton class**:
  - At **any** given **time**, only **one instance**.
  - **Private constructor**
  - Does **not** use **new** keyword, uses **getInstance()** method(by convention).
  - Method **returns an object** to the class.

```

public class Calender {

    private static Calender instance = null;

    private Calender() {

    }

    public Calender getInstance() {

        if (instance == null) {

            instance = new Calender();

        }

        return instance;

    }

}

```

- **Abstraction: Hiding details, showing functionality.**
- **Encapsulation: Code and functions in a single unit.**

- **new** is used to **allocate memory** at **runtime**.
- **Anonymous objects** are **nameless** objects. There are **no references** to these objects.

```
new Person("Name").sayHello();
```

# Exception Handling

- **Exception:** On **execution**, **disrupts** flow, **unwanted**, **unexpected** event.
- **Error:** On **execution**, **problem** on **system**.
- Exceptions and errors are **sub-classes** of **Throwable** class.
- **Exceptions:**
  - **Checked**
  - **Unchecked**
- **Errors:** Virtual Machine errors, Assertion error, ...
- **Checked Exceptions:** IO Exceptions, Compile time Exceptions ...
- **Unchecked Exceptions:** Runtime Exception, NullPointerException ...
- **Default exception handling:**
  1. Method **creates** an **Exception object** and **sends** it to **JVM**.
  2. **Exception** has **name**, **explanation** and **current program status**.
  3. This process is called exception **throwing**.
  4. In every exception **raise**, there is a list called **Call Stack** which lists all methods. It is **important** to write **catch** blocks with respect to **hierarchy**.
  5. In an **exception raise**, **run-time** system searches for a **method** which can **handle** the **exception** on the **call stack**.
  6. This code block is called **exception handler**.
  7. If run-time system can **find** a **related exception handler**, it **transfers** exception **to method**.
  8. If run-time system can **not find** a **related exception handler**, it **transfers** exception **to default exception handler**.
  9. Default exception handler **prints** exception **information** and **ends program abnormally**.
  10. **Code** block that **could raise an exception**, **should** be written in a **try-catch** block.

```
try {
    // Code that may raise an exception
} catch (Exception e) {
    e.printStackTrace();
}
```

11. **Inside** of a **try** block, **exception raises**, try block **throws exception**.
12. Thrown exception is tried to **catch** from **one of the catch blocks**.
13. **System exceptions** are automatically **thrown** by **JRE**.
14. You may **throw exception manually**.

```
...
if (param == null) {
    throw NullPointerException;
}
...
```

15. Every **throwable** exception should be **written** on **method definition**.

- Exception messages:

- `java.lang.Throwable.printStackTrace()`
    - `toString()`
    - `getMessage()`
  - Some of the **important** built-in **exceptions**:
    - `ArithmeticException`
    - `ArrayIndexOutOfBoundsException`
    - `ClassNotFoundException`
    - `FileNotFoundException`
    - `IOException`
    - `InterruptedException`
    - `NoSuchFieldException`
    - `NoSuchMethodException`
    - `NullPointerException`
    - `NumberFormatException`
    - `RuntimeException`
    - `StringIndexOutOfBoundsException`
  - **Integer** div by **0** will throw `java.lang.ArithmeticException: / by zero exception`
- 

# Garbage Collection

---

- **Mark and Sweep - Shuffling**
  - **GC time:** **Decreases** with **increase** of **dead object number**, **increases** with **increase** of **live object number**.
  - **Wrapper class:** Primitive data type --> Object
  - **Wrapper classes helps** to **use primitives** with **Collections**. (`ArrayList`, `Vector`)
  - **ArrayList** and **Vector**, both **extends AbstractList** and **implements List** interface.
  - Objects on **heap memory** could refer to themselves. This would cause a **loop**. This is called **island of isolation**.
  - When an **object** is **created** its **sign bit** is set to **false**.
  - On **marking phase**, all **reachable** objects' sign bit is set to **true**. To reach, **GC** uses **DFS**.
  - On **sweep phase**, all objects with **false sign bit** is **cleaned** from **heap memory**.
  - On every **method call**, method goes to **stack frame**. When it is **popped**, all **members die(F)**. If there is **any object** created **inside** method, it will **die(F)**.
  - If a variable **keeps reference** to an **object** and programmer **assigns another reference to another object**, first object will be **unreachable(F)**.
  - If all references to an object is **null**, object will be **unreachable(F)**.
  - Anonymous object's id is **not** stored. So it will be **eligible(F)** for **GC**.
  - **Wrapping** of primitive data types are important for **multithreading sync**.
  - **Wrapper objects** are **immutable**. On variable value **changings**, in background, a **new object** is created, object will be **unboxed**, arithmetic **operation** will be done, **new value** will be **boxed**, new object **reference** will be **assigned** to the **object**.
- 

# JVM - JRE - JDK

---

- Java is **architecture-neutral**. There are **no implementation dependent** features.
- Having **semicolon** at the **end** of a **class** definition is **optional**.
- JVM Runtime Operations:
  - **Class** file is **loaded** by **ClassLoader**.
  - **Bytecode verifier** checks for illegal operations.

- **Interpreter** reads **bytecode** and executes instructions on hardware.
  - **JVM, JRE and JDK** are platform **dependent**. Because every **architecture** needs **different configurations**.
  - Language itself is platform independent.
  - **JDK = JRE + Dev Tools**
  - **JRE = JVM + Libraries**
  - **JVM = Classloader + Memory Areas + Execution Engine + Native Method Interface + Native Libraries**
- 

# JVM

---

- **Classloader**: Loads class files to JVM.
  1. **Bootstrap Classloader**:
    - Loads **rt.jar** file
    - This file contains:
      - **java.lang**
      - **java.net**
      - **java.util**
      - **java.io**
      - **java.sql**
  2. **Extension Classloader**:
    - Loads **\$JAVA\_HOME/jre/lib/ext**
  3. **System Classloader**:
    - Loads class files from **classpath**.
    - Default: current directory
- **Memory areas** allocated by JVM:
  1. **Class Area**:
    - **Stores class structures**
    - Holds constants, member fields and instance method datas.
    - Method codes
  2. **Heap Memory**:
    - **Runtime** data area
    - **Objects** are **allocated** here.
  3. **Stack**:
    - Stores **frames**.
    - Holds **local variables**.
    - Method invocation (**Assembly**)
    - **Return** value
    - Each **thread** has private stack.
  4. **PC Register**:
    - Same as **Instructor Pointer(IP)** in **80x86 Assembly**
  5. **Native Method Stack**:
    - **Contains** all **native(built-in) methods** used in program.
- **Execution Engine = Virtual Processor + Interpreter + JIT**
- **JIT, compiles** bytecode in **blocks**(similar functionality).



- **Native Method Interface: Interface** for **other programs** written in another language.
- 

- **Writing the state of an object into a byte stream** is called **serialization**.
  - Serialization mostly used on **networking**.
  - **java.io.Serializable** is an **interface** used to **mark a class** to provide capabilities.
  - **java.lang.String** and **Wrapper classes** implements **java.io.Serializable**.
  - Only **objects** which is an instance of a class that **implements java.io.Serializable** can be **written to streams**.
  - **transient** is used in **serialization**. If you **define** any data as **transient**, it **won't** be **serialized**.
- 

- Types of **inheritance**:

- **Single** (supported)

```
class A {  
  
}  
  
class B extends A {  
  
}
```

- **Multilevel** (supported)

```
class A {  
  
}  
  
class B extends A {  
  
}  
  
class C extends B {  
  
}
```

- **Hierarchical** (supported)

```
class A {  
  
}  
  
class B extends A {  
  
}  
  
class C extends A {  
  
}
```

- **Multiple** (**not** supported)
- **Hybrid** (also known as **Diamond**) (**not** supported)
- **Call to an overridden method resolves** at **runtime** (not compile-time). This is called **runtime polymorphism(Dynamic Method Dispatch)**
- An **abstract class** can have:
  - **Abstract** and **non-abstract** methods
  - **Static** methods
  - **Final** methods
- **static** keyword may be used with:
  - **Variables**

- **Methods**
- **Blocks**
- **Nested Classes**
- **Static variables** are **created** only **once** during the **loading of** related **class**.
- **Static methods** belongs to all objects of that class.
- You may **call a static method without** the need of creating **an instance** of the class.
- **You can not refer to a non-static context from a static context.**
- You **can not** use **this** and **super** in static context.
- **Static blocks:**
  - Used to **initialize static member variables**.
  - **Executed** at the **classloading**.