**Exception propagation:** An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible “something” to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the **call stack** and the method of searching is **Exception Propagation.**

**Exception Propagation in Unchecked Exceptions**

When an exception happens, Propagation is a process in which the exception is being dropped from to the top to the bottom of the stack. If not caught once, the exception again drops down to the previous method and so on until it gets caught or until it reach the very bottom of the call stack. This is called exception propagation and this happens in case of Unchecked Exceptions.

|  |
| --- |
| **public** **class** Unchecked\_Exceptions {  **public** **static** **void** main(String[] args) {  Unchecked\_Exceptions obj = **new** Unchecked\_Exceptions();  obj.p();  System.***out***.println ("Normal flow...");  }  **void** p() {  **try** {  n();  } **catch** (Exception e) {  System.***out***.println("Exception handled");  }  }  **void** n() {  m();  }  **void** m() {  **int** data = 50 / 0;  }  } |

**Exception Propagation in Checked Exceptions**

Unlike Unchecked Exceptions, the propagation of exception **does not happen** in case of Checked Exception and it’s mandatory to use [throw keyword](https://www.geeksforgeeks.org/throw-throws-java/) here. Only unchecked exceptions are propagated.**Checked exceptions throw compilation error.**

|  |
| --- |
| **public** **class** Checked\_Exceptions {  **public** **static** **void** main(String[] args) {  Checked\_Exceptions obj = **new** Checked\_Exceptions();  obj.p();  System.***out***.println ("normal flow...");  }  **void** p() {  **try** {  n();  } **catch** (Exception e) {  System.***out***.println("exception handled");  }  }  **void** n() **throws** IOException {  m();  }  **void** m() **throws** IOException {  **throw** **new** IOException("device error");  }  } |

# SerialVersionUID in Java

The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

**Why so we use SerialVersionUID:**SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded.

**Serialization:**At the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system.

**Deserialization:** At the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise we will get Runtime Exception saying **InvalidClassException**. This unique Identifier is nothing but **SerialVersionUID**.

# Verification in Java (JVM)

After the class loader in the [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)loads the byte code of .class file to the machine the Bytecode is first checked for validity by the verifier and this process is called as **verification**. The verifier perform as much checking as possible at the Linking so that expensive operation performed by the interpreter at the run time can be eliminated. It enhances the performances of the interpreter.

**Some of the checks that verifier performs:**

* Uninitialized Variables
* Access rules for private data and methods are not violated.
* Method calls match the object Reference.
* There are no operand stack overflows or underflows.
* The arguments to all the Java Virtual Machine instructions are of valid types.
* Ensuring that final classes are not sub classed and that final methods are not overridden
* Checking that all field references and method references have valid names, valid classes, and a valid type descriptor. ([source](https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10))

If any of these checks fails JVM throws a “java.lang.VerifyError” error. However we can disable these checks using

java -noverify VerifyGeekFile

# Final vs Immutability in Java

**Final:**In Java, [final](https://www.geeksforgeeks.org/final-keyword-java/) is a modifier which is used for class, method and variable also. When a variable is declared with final keyword, its value can’t be modified, essentially, a constant.

[**Immutability**](https://www.geeksforgeeks.org/create-immutable-class-java/)**:**In simple terms, immutability means unchanging over time or unable to be changed. In Java, we know that String objects are immutable means we can’t change anything to the existing String objects.

**Differences between final and immutability**

* Final means that you can’t change the object’s reference to point to another reference or another object, but you can still mutate its state (using setter methods e.g.). Whereas immutable means that the object’s actual value can’t be changed, but you can change its reference to another one.
* Final modifier is applicable for variable but not for objects, whereas immutability applicable for object but not for variables.
* By declaring a reference variable as final, we won’t get any immutability nature, Even though reference variable is final. We can perform any type of change in the corresponding Object. But we can’t perform reassignment for that variable.
* Final ensures that the address of the object remains the same whereas the Immutable suggests that we can’t change the state of the object once created.

# Java 8 Predicate with Examples

[Functional Interface](https://www.geeksforgeeks.org/functional-interfaces-java/) is an Interface which allows only one Abstract method within the Interface scope. There are some predefined functional interface in Java like Predicate, consumer, supplier etc.

The Functional Interface **PREDICATE** is defined in the *java.util.Function package*.It improves manageability of code, helps in unit-testing them separately, and contain some methods like:

1. **isEqual(Object targetRef) :**Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

**static Predicate isEqual(Object targetRef)**

Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

**T :** the type of arguments to the predicate

**Parameters:**

**targetRef :** the object reference with which to compare for equality, which may be null

**Returns:** a predicate that tests if two arguments are equal according to Objects.equals(Object, Object)

1. **and(Predicate other) :**Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

**default Predicate and(Predicate other)**

Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.

**Parameters:**

other: a predicate that will be logically-ANDed with this predicate

Returns : a composed predicate that represents the short-circuiting logical AND of this predicate and the other predicate

Throws: NullPointerException - if other is null

1. **negate() :** Returns a predicate that represents the logical negation of this predicate.

**default Predicate negate()**

Returns:a predicate that represents the logical negation of this predicate

1. **or(Predicate other) :** Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.

**default Predicate or(Predicate other)**

Parameters:

other : a predicate that will be logically-ORed with this predicate

Returns: a composed predicate that represents the short-circuiting logical OR of this predicate and the other predicate

Throws : NullPointerException - if other is null

1. **test(T t) :** Evaluates this predicate on the given argument.boolean test(T t)

**test(T t)**

Parameters:

t - the input argument

Returns:true if the input argument matches the predicate, otherwise false

# POJI in Java

**POJI:** stands for Plain Old Java Interface. A POJI is an ordinary interface without any specialties. The interfaces that do not extend from technology/framework specific interfaces. For example all user defined interfaces are POJI and an interface that inherits from AppletInitializer of Java Beans is not POJI.

Examples:

|  |
| --- |
| // A POJI interface  interface GFG {    public void method1();  }    interface Geeks extends GFG {    public void method2();  } |
| interface GFG extends java.io.Serializable { } |

// Not a POJI Interface

interface GFG1 extends java.rmi.Remote {}

// Not a POJI Interface

interface GFG2 extends java.beans.AppletInitializer {}

# CLASSPATH in Java

the CLASSPATH environment variable if responsible for. While programming in Java, we many times use import statements. An example is:

import org.company.Menu

What does this import means? It makes the Menu class available in the package org.company to our current class. Such that when we call

Menu menu = new Menu()

The [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) knows where to find the class ***Menu***. Now, how will the JVM know this location? It is impractical for it to go through every folder on your system and search for it. Thus, using the CLASSPATH variable we provide it the place where we want it to look. We put directories and jars in the CLASSPATH variable.  
Let’s say the above package resides in the directory *dir*. The complete path of the *Menu* class file would be *dir/org/company/Menu.* We’ll specify only the directory *dir* in our classpath variable, as rest information regarding the path is provided by the import statements. Similar for jar, if you create a jar and mention its path in the variable, the VM will look inside the jar file and find the class.