Branch: master ▾ c2c2017 / Session10 /

Create new file Find file History

havanagrawal committed on GitHub Fix minor typo — Latest commit e678f45 10 hours ago

..		
gc	Add Garbage Collection worksheet to Session10	11 hours ago
problems	Add Trigram/word prediction problem to Session 10	18 hours ago
README.md	Fix minor typo	10 hours ago
exception_hierarchy.png	Add missing arrows for IOException classes	10 hours ago

README.md

Session 10

Table of Contents

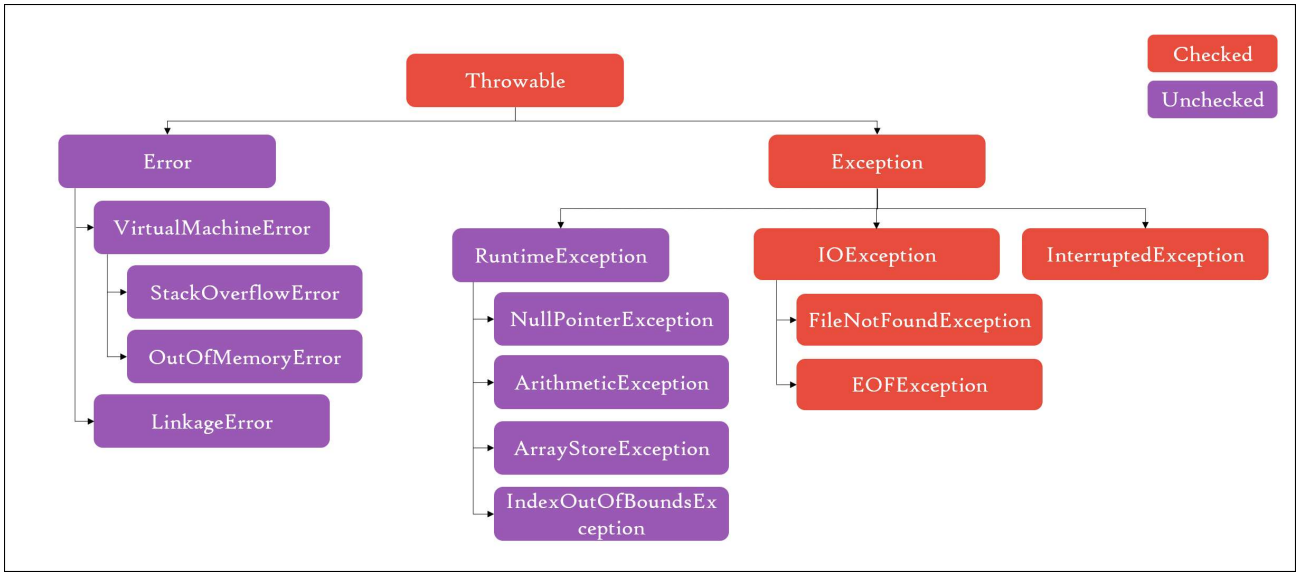
Exception Handling

The Java programming language uses exceptions to handle errors and other exceptional events.

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

The Exception Hierarchy

All exception and error implementations occur under the Throwable class in Java.



Checked and Unchecked Exceptions

Exceptions in Java have been partitioned into two types; checked and unchecked exceptions. Checked exceptions are checked by the *compiler*, in that if a method throws a checked exception, then it must either handle it using a try-catch block, or explicitly declare it to be thrown using the throws keyword.

Unchecked exceptions are not checked by the compiler, and so the code that raises such an exception is not bound by any additional constraints.

Everything under the RuntimeException and Error classes are unchecked. Every other exception is checked.

While the intent behind categorizing these two was good, it is currently considered an overhead. There is a large set of programmers in the world, who believe that Java should have only unchecked exceptions.

In general, the guideline for deciding whether an exception you are making should be checked or unchecked is:

1. If the client using your API/library/code can recover from an exception thrown by your code, it should be checked. For instance, if your library allows a connection to Instagram, but limits the number of connections, and throws a "TooManyConnectionsException", the client/user can close an existing connection (in a catch block), and retry.
2. If there is no possibility of recovery, use an unchecked exception, i.e. extend from RuntimeException. For instance, if your aforementioned library fails because the Instagram server is down, there is hardly anything that the user can do to recover from this. Unchecked exceptions should also be used to indicate incorrect usage of an API. For example, if the client code passes null arguments to your authenticate method, or something similar, you can, and should throw an IllegalArgumentException, which is unchecked.

throw and throws

The throw keyword is used to, unsurprisingly, throw an exception. Continuing the example from above:

```
class InstagramClient {
  ...

  public void authenticate(String clientKey, String clientPassword) {
    if (clientKey == null) {
      throw new IllegalArgumentException("clientKey cannot be null!");
    }

    if (clientPassword == null) {
      throw new IllegalArgumentException("clientPassword cannot be null!");
    }
    ...
  }
}
```

The throws keyword is used in a method declaration, to declare to the caller that this method throws a checked exception. You have already seen an

example of this:

```
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    String s = br.readLine();
}
```

Since IOException is a checked exception, it must be declared to be thrown by the method.

If you check the documentation for [readLine](https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html#readLine() method of BufferedReader, you'll see that it declares that it throws IOException.

try-catch-finally

Since we've been reading about things being thrown around, there is an obvious construct to "catch" these exceptions.

The try-catch block allows you to "try" to execute a piece of code, and if an exception is thrown, you "catch" it, and either try to recover from it, or fail gracefully.

The following is an overused, yet simple example for using try-catch

```
class ArithmeticExceptionExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();

        int c = divide(a, b);
    }

    public int divide(int a, int b) {
        int x;
        try {
            // b could be 0!
            x = a / b;
        }
        catch (ArithmeticException ae) {
            System.out.println(ae);
            return 0;   // Does not make sense to return 0, but what can you do :P
        }
        return x;
    }
}
```

The finally block is ALWAYS executed, regardless of whether a try is successful or not.

For example:

```
try {
    System.out.println("In try block");
    throw new IOException("Good example for try-catch not found!");
}
catch (IOException ioe) {
    System.out.println("In catch block");
}
finally {
    System.out.println("In finally block");
}
```

will print:

```
In try block
In catch block
In finally block
```

and

```
try {
    System.out.println("In try block");
}
catch (IOException ioe) {
    System.out.println("In catch block");
}
finally {
    System.out.println("In finally block");
}
```

will print:

```
In try block
In finally block
```

Remember that the finally block is **ALWAYS** executed.
When is it executed? **ALWAYS**.

Variants of try-catch-finally

There are several possible variations of try-catch-finally, not all of which are correct. The following sections attempts to cover all of them, clearly specifying whether it is valid or invalid.

Valid

1. Simple try-catch

```
try {
    // some code that throws IOException
}
catch (IOException e) {
    // some code
}
```

2. Simple try-catch-finally

```
try {
    // some code that throws IOException
}
catch (IOException e) {
    // some code
}
finally {
    // some code
}
```

3. Nested try inside a catch (or finally) block

```
try {
    // some code that throws IOException
}
catch (IOException e) {
    try {
```

```
        // some code
    }
    catch (DataFormatException) {
        // some code
    }
}
```

4. Cascading catches

```
try {
    // some code that throws IOException
}
catch (IOException ioe) {
    // some code
}
catch (ArithmeticException ae) {
    // some code
}
```

5. Try-finally with no catches

```
try {
    // some code
}
finally {

}
```

Invalid

1. Just try

```
try {
    // some code
}
```

2. Catching *checked* exceptions that cannot be thrown by the code in the try block

```
try {
    // some code that never throws an IOException
}
catch (IOException ioe) {

}
```

3. Catching exceptions higher in the hierarchy before sub-class exceptions in a cascading catch

```
try {
    // some code
}
catch (Exception ioe) {

}
catch (IOException ioe) {
    // This is unreachable code!
}
```

Call Stack propagation

Q: What happens when an exception is thrown, and you don't catch it?
A: The exception will propagate up through the call stack, till either someone catches it, or it gets thrown out of main, in which case the JVM stops the program and prints out the stack trace to the console.

It is important to remember that for a checked exception to get propagated through the call stack, *every* method on the stack will have had to have declared throwing that exception.

Some "Exceptional" Cases

1. What do you think is returned from this function? Try to execute it and find out!

```
public static int whatThe() {
   try {
      return 10;
   }
   finally {
      return 20;
   }
}
```

2. The `finally` block **ALWAYS** executes, *unless* the JVM itself shuts down, one way of doing which is `System.exit()`. **DO NOT EVER CALL System.exit, it is a foul habit!**

Bonus: A Lame Joke

Garbage Collection

When objects on the heap are no longer accessible by a reference on the stack, they are cleaned up by the garbage collector.

The Garbage Collector

The Garbage Collector is a vast topic in itself, with varying implementations, each with its own performance and efficiency characteristics.

For now, it is sufficient to understand that the GC simply performs a DFS to look for live objects on the heap, starting from root references (on the stack), and then marks everything else as garbage collectible.

Note that we will always say that an object is "eligible" for garbage collection, and never "is being garbage collected", *because you cannot guarantee when the garbage collector runs.*

The `finalize` method

The `finalize` method is used to perform any clean up operations, when the instance is being garbage collected.

A common misconception is that it is synonymous to a destructor in C++. **It is not.**

As usual, one of the best places to understand such core functionalities is the JavaDoc. The `finalize` method resides in the Object class, and so every class inherits the `finalize` method.

It is rare to see this method overridden by base classes.

Heaps

Assignments

HackerRank

1. https://www.hackerrank.com/challenges/java-exception-handling-try-catch
2. https://www.hackerrank.com/challenges/java-exception-handling

Miscellaneous