# Exception handling and annotations

- Exception handling and annotations
  - Exceptions
    - Types of Exceptions
    - Exception Hierarchy
    - Exception Handling
      - The throw keyword
      - The throws keyword
      - The try and catch keywords
      - The finally keyword
      - The finalize method
  - | final is executed only when we call it.| finally block executes as soon as the execution of try-catch block is completed without depending on the exception. | finalize method is executed just before the object is destroyed. |
    - The golden Rules of Exception Handling
  - Annotations
    - Built-in Annotations
    - Categories of Annotations

## Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. In Java, an exception is an object that wraps an error event that occurred within a method and contains:

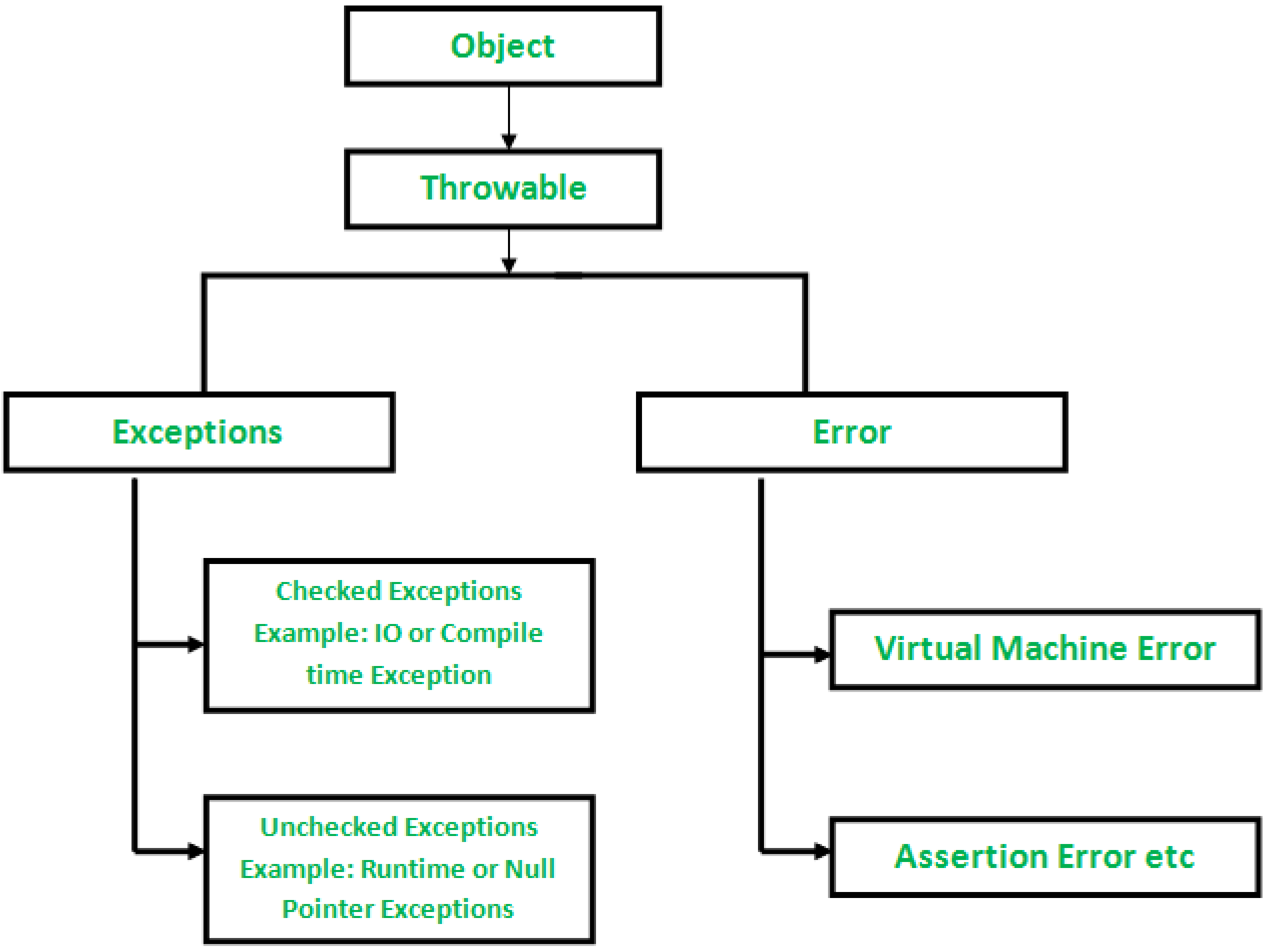
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information about the error

### Types of Exceptions

There are two types of exceptions:

- Checked exceptions are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. Example: IOException, SQLException, etc. Checked exceptions are also called as compile time exceptions.
- Unchecked exceptions are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers either handle them or let them propagate up. Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are also called as runtime exceptions.

### Exception Hierarchy


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class, there is another subclass called Error which is derived from the Throwable class. Errors are not normally trapped by the Java programs. These conditions normally happen in case of severe failures, which are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally programs cannot recover from errors.

## Exception Handling

#### The throw keyword

The throw keyword is used to explicitly throw an exception. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

```
public int getPlayerScore(String playerFile) {
    if (playerFile == null) {
        throw new IllegalArgumentException("Player file cannot be null");
    }
    Scanner contents = new Scanner(new File(playerFile));
    return Integer.parseInt(contents.nextLine());
}
```

Since IllegalArgumentException is an unchecked exception, we don't need to declare it in the method signature. If we want to throw a checked exception, we need to declare it in the method signature.

#### The throws keyword

The simplest way to "handle" an exception is to rethrow it. This is done using the throws statement. The general form of throw is shown here:

```
public int getPlayerScore(String playerFile) throws FileNotFoundException
{

    Scanner contents = new Scanner(new File(playerFile));
    return Integer.parseInt(contents.nextLine());
}
```

The throws clause appears at the end of a method's signature. It informs the caller that the method might throw an exception. If it does, then the caller must either catch the exception or rethrow it. In the preceding example, the throws clause indicates that getPlayerScore( ) might throw a FileNotFoundException. Thus, any code that calls getPlayerScore( ) must either catch the exception or declare that it also can throw it.

#### The try and catch keywords

The try statement allows you to define a block of code to be tested for errors while it is being executed. The catch statement allows you to define a block of code to be executed, if an error occurs in the try block. The try and catch keywords come in pairs:

```
public int getPlayerScore(String playerFile) {
    try {
        Scanner contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } catch ( FileNotFoundException noFile ) {
        logger.warn("File not found, resetting score.");
        return 0;
    }
}
```

#### The finally keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
public int getPlayerScore(String playerFile)
  throws FileNotFoundException {
    Scanner contents = null;
    try {
        contents = new Scanner(new File(playerFile));
        return Integer.parseInt(contents.nextLine());
    } finally {
        if (contents != null) {
            contents.close();
        }
    }
}
```

**The finalize method**

The finalize( ) method is called just before an object is destroyed and is used to handle any cleanup processing. This method is defined by Object, so all classes have finalize( ) defined. However, the default version of finalize( ) does nothing. finalize( ) has this general form:

```
protected void finalize() {
    // finalization code here
}
```

The finalize( ) method is never invoked more than once for any given object. The finalize( ) method can be overridden. The finalize( ) method is not automatically called for an object when the program exits — this is because the Java runtime environment exits when the program exits, not when no references to objects exist.

| Final | Finally | Finalize |
|---|---|---|
| final is a keyword and access modifier in Java. | finally block is used in Java Exception Handling to execute the mandatory piece of code after try-catch blocks. | finalize() is the method in Java. |
| final non-access modifier is used to apply restrictions on the variables, methods, and classes. | finally block executes whether an exception occurs or not. | It is primarily used to close resources. finalize() method is used to perform clean-up processing just before an object is garbage collected. |
| It is used with variables, methods, and classes. | It is with the try-catch block in exception handling. | It is used with objects. |
| Once declared, the final variable becomes constant and can't be modified. | finally block cleans up all the resources used in the try block. | finalize method performs the cleaning with respect to the object before its destruction. |

| Final | Finally | Finalize |
|---|---|---|
| final is executed only when we call it. | finally block executes as soon as the execution of try-catch block is completed without depending on the exception. | finalize method is executed just before the object is destroyed. |

---

### The golden Rules of Exception Handling

1. Never swallow an exception - Swallowing an exception means that you catch it and do nothing with it. This is a bad practice because it means that you are ignoring the fact that an error occurred, which could cause your program to behave in unexpected ways and hide bugs.

```java
try {
    // do something
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```

2. Never catch a generic exception - Catching a generic exception means that you are catching all exceptions, including runtime exceptions. This is a bad practice because it means that you are not handling exceptions in a meaningful way. You should always catch specific exceptions and handle them appropriately.

```java
try {
    // do something
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```

3. Never throw a generic exception - Throwing a generic exception means that you are throwing all exceptions, including runtime exceptions. This is a bad practice because it means that you are not handling exceptions in a meaningful way. You should always throw specific exceptions and handle them appropriately.

```java
public void doSomething() throws Exception {
    // do something
}
```

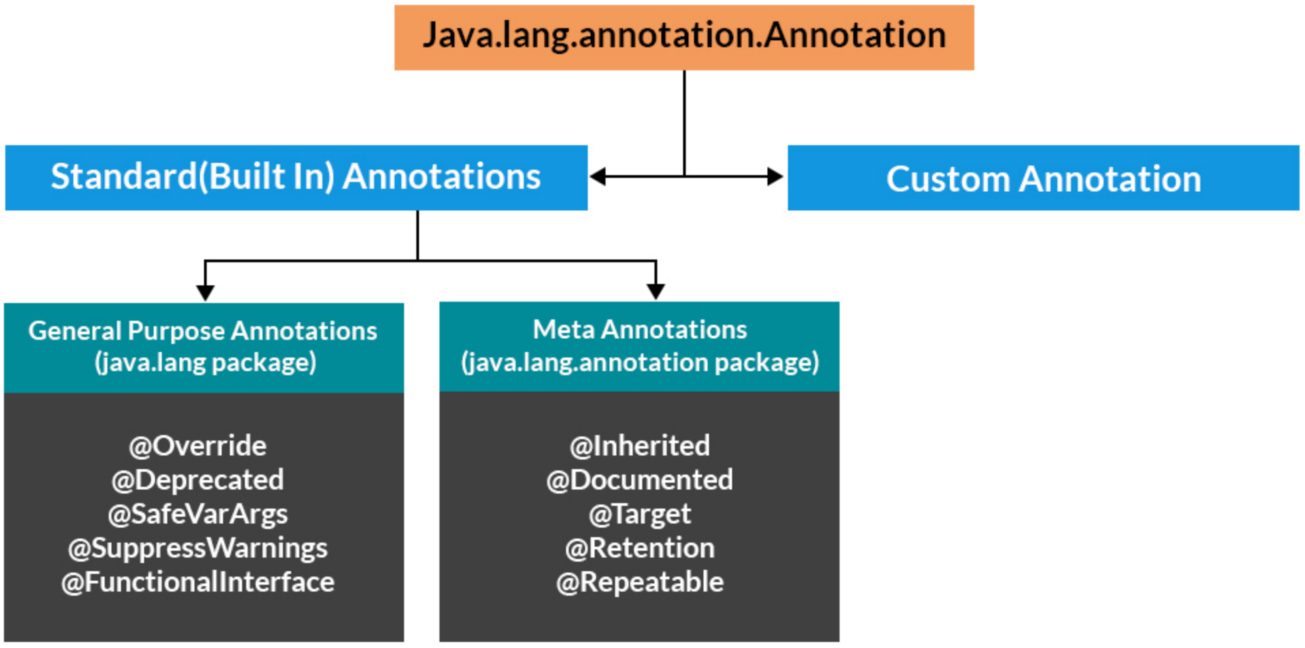
---

## Annotations

Annotations are a form of metadata that provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate as they do not change the semantics of the compiled program. However, they are also not pure comments as they can be processed by compiler tools in some way. Annotations were added to the Java language in JDK 5.0.

Annotations are used for the following purposes:

- Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
- Compile-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.


Java.lang.annotation.Annotation
Standard(Built In) Annotations
Custom Annotation
General Purpose Annotations (java.lang package)
@Override
@Deprecated
@SafeVarArgs
@SuppressWarnings
@FunctionalInterface
Meta Annotations (java.lang.annotation package)
@Inherited
@Documented
@Target
@Retention
@Repeatable

### Built-in Annotations

Java provides a set of built-in annotations that are available for general use. These annotations are used by the compiler and other tools. Built-in annotations can be divided into the following categories:

- General-purpose annotations - `java.lang.*` - These annotations are used by the compiler to detect errors, suppress warnings and enable or disable compile-time and deployment-time processing of annotations.
- Meta-annotations - `java.lang.annotation.*` - These annotations are used to create other annotations.

### Categories of Annotations

Annotations can be divided into the following categories:

- Marker annotations - These annotations contain no members. They simply mark the declaration to which they apply with some additional information. Example: `@Override`, `@Deprecated`, `@SuppressWarnings`.
- Single-value annotations - These annotations contain a single member. Example: `@SuppressWarnings("unchecked")`.
- Full annotations - These annotations contain multiple members. Example: `@Author(name = "John Doe", date = "3/17/2002")`.

There are other sub-categories such as type and repeating annotations.

---