**Module-4: Exception Handling**

**What is Exception?**

In Java, Exception is an **unwanted** or **unexpected** **event**, which occurs during the **execution of a program**, i.e. at run time, that **disrupts the normal flow** of the program’s instructions.

It is an object that is thrown at runtime when an unexpected situation (error) occurs.

**Exception Handling:**

Exception handling in Java is a mechanism to **manage runtime errors**, ensuring that the **normal flow** of the application is maintained. It allows a program to deal with unexpected events or errors without crashing.

**What is Error:**

**Exceptional conditions** that the application cannot normally **predict** or **recover** from. They are also ignored throughout the compilation process.

Errors represent **irrecoverable conditions** such as **Java virtual machine (JVM) running out of memory**, **memory leaks**, **stack overflow errors**, **library incompatibility**, **infinite recursion**, etc. Errors are usually **beyond the control of the programmer**, and we should not try to handle errors.

**Some examples of errors:**

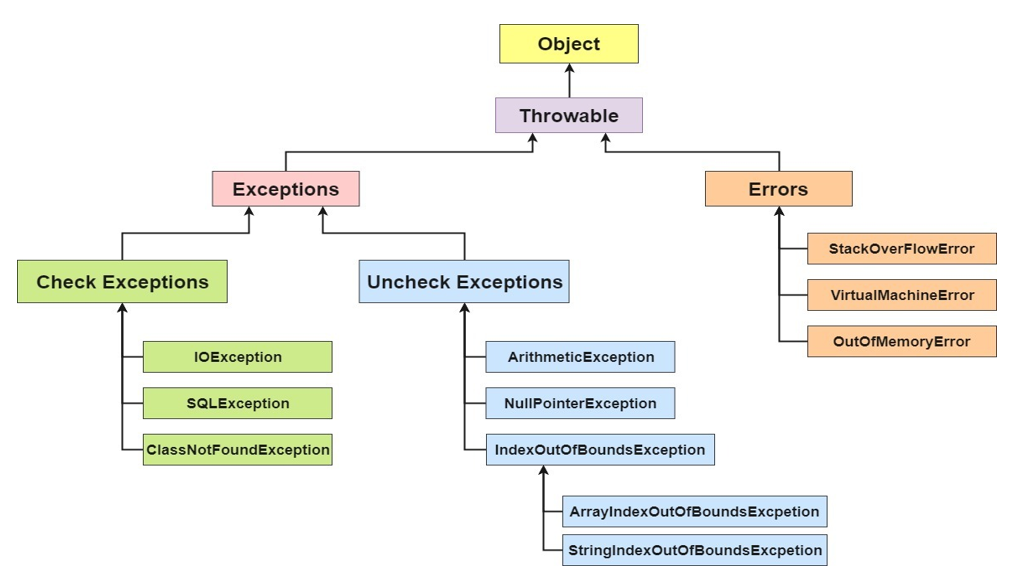
* OutOfMemoryError
* VirtualMachineErrors
* AssertionError

**Diff between Error and Exception:**

1. **Error:** Represents serious, unrecoverable issues that occur in the runtime environment. Errors are usually external to the application and indicate a problem that the application **cannot handle** or **recover** from (e.g., memory exhaustion, system crashes).
2. **Exception:** Represents conditions that a program might want to catch and handle. These are usually conditions that arise because of **logical** or **runtime** issues in the application (e.g., invalid user input, I/O problems, divide by zero).

**Exception Hierarchy:**

All exception and error types are **subclasses** of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).



1. **Checked or compiled-time exceptions:**

Arise when something goes wrong in your code but is **potentially recoverable**. Checked at **compilation time**.

**Some checked Exceptions include:**

* **SQLException:** It occurs due to various database-related issues
* **IOException:** Occurs when an input-output operation fails or is interrupted.
* **ClassNotFoundException:** When The class is not available in the classpath or the class file is missing.
* **FileNotfound Exception:** When The specified file does not exist

**Handling:**

* Must be **caught** or **declared** because it’s a checked exception.
* Usually handled in a **try-catch block** where database queries are executed.

**2. Unchecked or Runtime Exceptions in Java:**

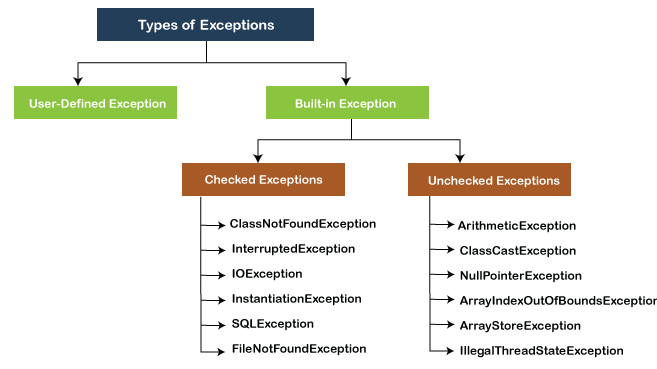
Unchecked exceptions are classes that inherit from **RuntimeException**. They are not tested at compile time, but rather during runtime.

**Some unchecked exceptions include:**

* **ArithmeticException:** This exception is thrown when an illegal arithmetic operation is attempted, such as division by zero
* **NullPointerException:** This exception is thrown when an application tries to use a null object reference where an object is required.
* **ArrayIndexOutOfBoundsException:** This exception is thrown when you try to access an array with an invalid index, such as a negative index or an index greater than or equal to the array size.
* **NumberFormatException:** This exception is thrown when an attempt is made to convert a string to a number
* **InputMismatchException:** This exception is thrown when an attempt is made to read data of the wrong type using a scanner. For example, if you try to read an integer but provide a non-numeric input.

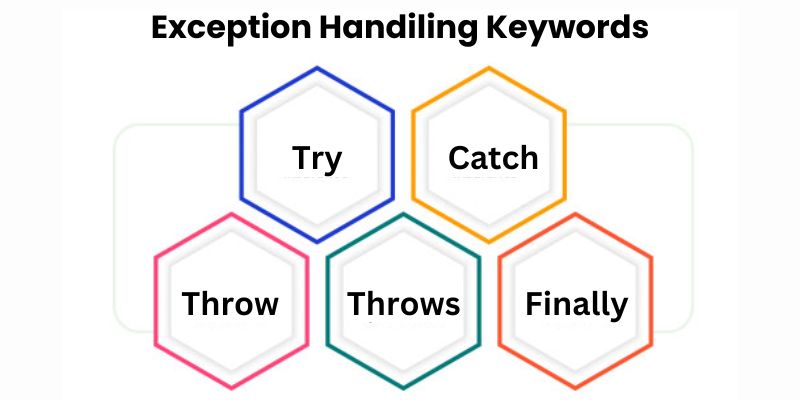
**Classification of Exceptions:**

In Java, exceptions are classified into two broad categories: **Checked Exceptions** and **Unchecked Exceptions**. These exceptions are further divided into different types based on their hierarchy and how they are handled.



**Important Keywords Related to Java Exception Handling:**

In Java, exception handling is done using **try**, **catch**, **throw**, **throws**, and **finally**. These constructs allow you to handle runtime errors gracefully.



1. **try: (Holds** Risky Code**)**

* The try block contains the code that **might throw an exception**. If an exception occurs, it **jumps to the catch block**.
* The try block is the block where the block of code that is needed to be **checked for exceptions** is placed. The try block is **followed by a catch or finally block, it cannot stand alone.**

**Example:**

A black screen with white text

Description automatically generated

**2. catch:**

* The catch block is used to **handle the exception thrown by the try block**. You can catch different types of exceptions by **chaining** multiple catch blocks. It is declared after the try block.

**Example:**

A computer code on a black background

Description automatically generated

**3. finally:**

* The finally block contains code that will **always execute** after the **try** and **catch** blocks, regardless of whether an **exception was thrown**. It is typically used for cleanup actions, like closing resources.
* Using the finally block we can **execute an important piece** of **code** because the finally block will be executed **regardless** of what the outcome is from the try block.

**Example:**

A computer code on a black background

Description automatically generated

**4. throw:**

* The throw keyword is used to explicitly throw an exception.
* Using the throw keyword we can **throw a predefined exception.**

**Example:**

A computer code with text on it

Description automatically generated

**5. throws:**

The throws keyword is used in **method signatures** to declare that a **method might throw one or more exceptions**. This is useful for checked exceptions.

**Example:**

A black screen with white text

Description automatically generated

**Creating own Exception sub-classes:**

In Java, you can create your own custom exception classes by **extending** the built-in **Exception** **class** (for checked exceptions) or **RuntimeException** class (for unchecked exceptions). Custom exceptions are useful when you want to **throw specific errors** related to your application's business logic.

**Steps to Create a Custom Exception Class:**

* **Extend** the Exception class (for checked exceptions) or RuntimeException class (for unchecked exceptions).
* Provide **constructors** to initialize the exception message and optionally pass other information like the cause of the exception.

**1. Creating a Custom Checked Exception:**

A checked exception must be declared in the method signature using the throws keyword and must be handled (caught) in the calling method.

Example:

A computer screen with text on it

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Explanation:**

**InvalidAgeException** is a custom checked exception that must be declared in the method signature (throws InvalidAgeException).

When the age is less than 18, an InvalidAgeException is thrown with a custom message.

The calling method (main()) must catch or declare this exception.

**2. Creating a Custom Unchecked Exception**

An unchecked exception is created by extending RuntimeException. Unlike checked exceptions, these do not need to be declared in the method signature or caught.

**Example:** Creating a Custom Unchecked Exception

A screen shot of a computer code

Description automatically generated

**Explanation:**

InsufficientFundsException is a custom unchecked exception (subclass of RuntimeException).

In this case, if the withdrawal amount exceeds the balance, an InsufficientFundsException is thrown.

This exception does not need to be declared in the method signature, nor does it need to be caught unless the developer chooses to.

A screen shot of a computer code

Description automatically generated