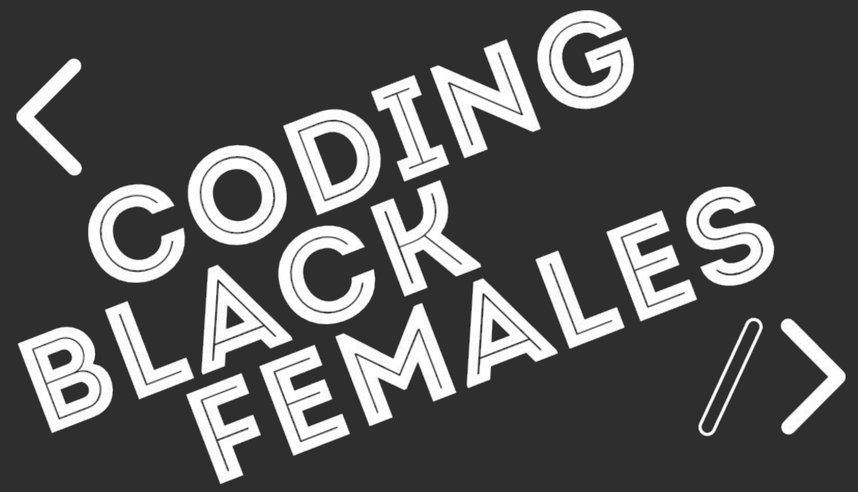


Exception Handling

Introduction to Java

CBF
ACADEMY

Session Outline

Errors & Exceptions

- Introduction
- Checked exceptions
- Unchecked exceptions
- Exception handling
- Creating custom exceptions

Learning Objectives

- Understand Java approach to error handling

CBF ACADEMY

Errors & Exceptions

CBF ACADEMY

Exception

An exception is an **unexpected event** that happens during program execution, and may cause it to terminate prematurely.

An exception can be caused by:

- invalid user input
- loss of connectivity
- opening an unavailable resource (file for example)
- code errors, ...

Exceptions can be caught and handled by the program.

Error

An error represent **irrecoverable conditions**, that are usually beyond our control, and we should not try to handle errors.

Example of errors:

- the JVM running out of memory
- memory leaks
- stack overflow errors
- infinite recursion, ...

In this course, we'll focus on **exceptions**.

Java Exceptions

When an exception occurs in a method, it creates an *object*. This is called an **exception object**. It contains:

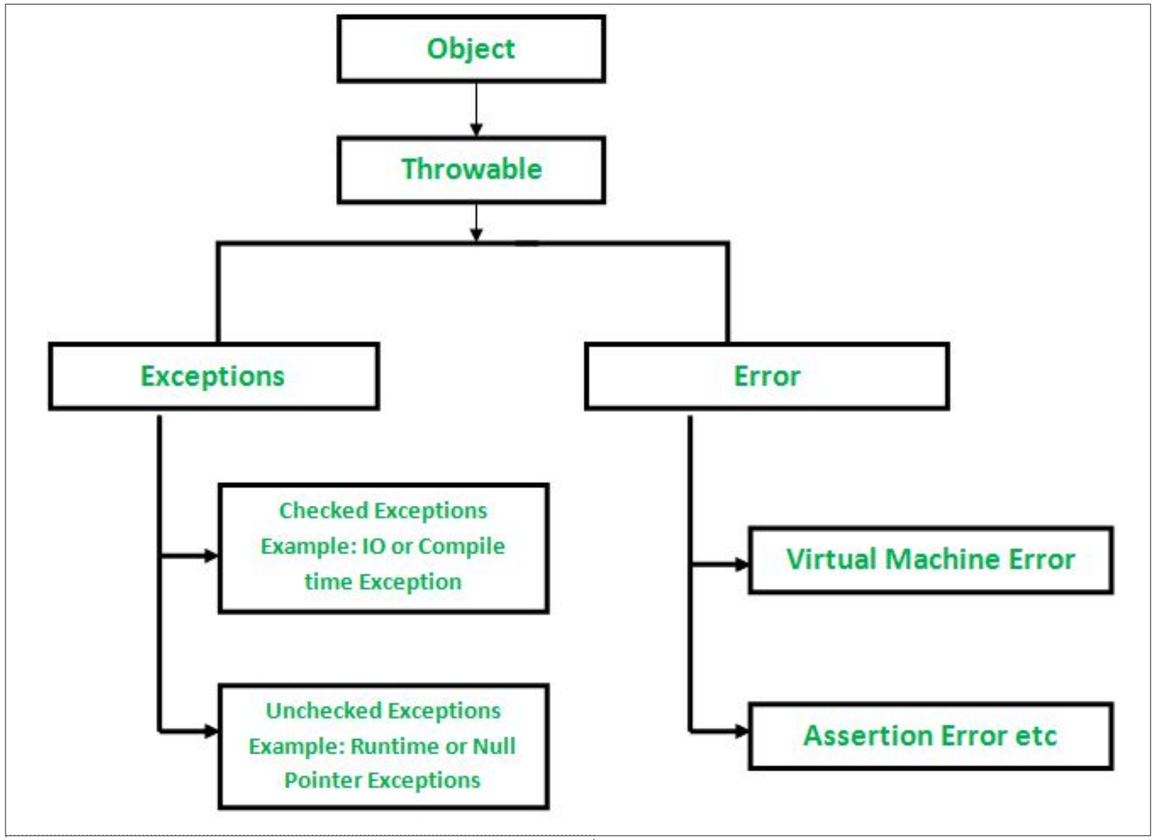
- the name of the exception
- the description of the exception
- the state of the program when the exception occurred

Any exception thrown from the **`main`** method (entry point of the program) is handled by the JVM.

CBF ACADEMY

Java Exceptions

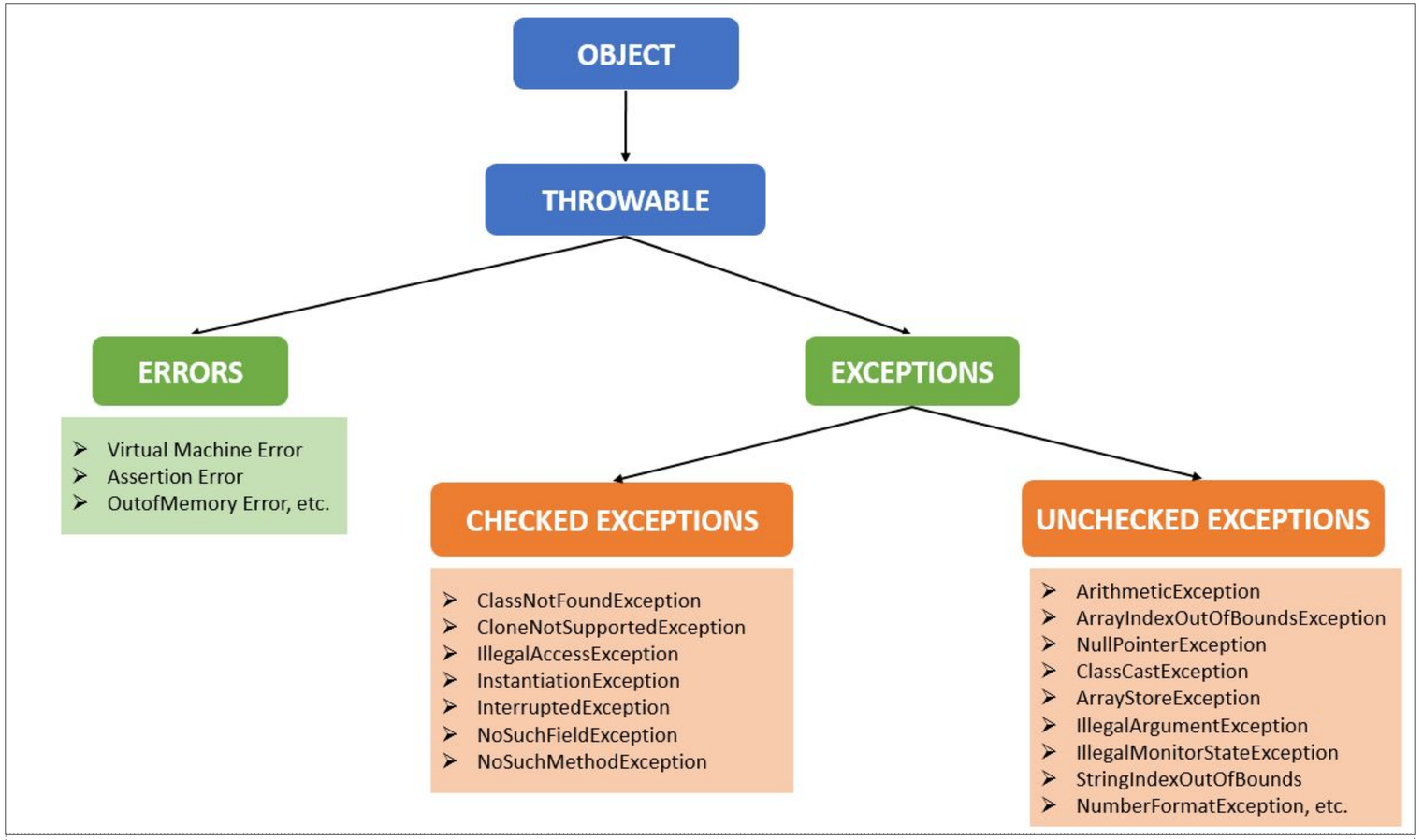
All exceptions in Java extend the **`Exception`**[1] class, which in turn extends the **`Throwable`**[1] class.



https://media.geeksforgeeks.org/wp-content/uploads/Exception-in-java1.png

Java Exceptions

Java exceptions are of two types: **checked** and **unchecked** exceptions.



https://lh6.googleusercontent.com/UuHJjdra7T1cHIuicA8GgpV9IPA0k7BkXmpgwXt68Eb_d29iEzBzKnPtX8r4Qs4a3GwT-_GW404bGv8F-VLhBfTer0hhbLnwtQwgC4g7GlCe1oL8naJ7c_j6as_po6DPfAQhxETOnQULknrZRA

CBF ACADEMY

Unchecked & Checked Exceptions

CBF ACADEMY

Unchecked Exceptions

Also known as **Runtime Exceptions,** they inherit from **RuntimeException**. They happen due to programming errors and aren't checked at compile-time, only at run-time.

Examples:

- **`ArithmeticException`** - due to a division by 0
- **`NullPointerException`** - due to illegal access on a variable prior to initialisation
- **`ArrayIndexOutOfBoundsException`** - due to an out of bounds array access

Checked Exceptions

They are checked at **compile-time**, and we are prompted to handle them when writing our code.

Examples:

- a **`FileNotFoundException`** - due to an attempt to open a file that might not exist
- a **`IllegalAccessException`** - due to an attempt to read past the end of a file for example

Example

unchecked

checked

```java
import java.io.File;
import java.util.Scanner;

public class Program {
    public static void main(String... args) {
        // This line might cause a NumberFormatException at runtime
        int first = Integer.parseInt(args[0]);

        // The compiler complains about an unhandled java.io.IOException
        File file = File.createTempFile( prefix: "prefix",  suffix: ".doc");

        // The compiler complains about an unhandled java.io.FileNotFoundException
        Scanner scanner = new Scanner(file);
    }
}
```


CBF ACADEMY

How are you feeling?

#intro-to-java-channel

🤯

I have no idea what you're talking about

✋

I have some questions but feel like I understand some things

👍

I feel comfortable with everything you've said

CBF ACADEMY

Handling Exceptions

CBF ACADEMY

Exceptions Handling

In Java, there are three constructs we can use to handle exceptions:

- the **`try ... catch`** block
- the **`finally`** block
- the use of the **`throw`** and **`throws`** keywords

Example.java

```java
import java.io.File;
import java.nio.file.FileSystemException;

public class Example {
    private final File file;

    public Example(File file) {
        this.file = file;
    }

    public String getName() throws FileSystemException {
        if (this.file == null) {
            throw new FileSystemException("No file");
        }

        return this.file.getName();
    }
}
```

Program.java

```java
import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystemException;

public class Program {
    public static void main(String... args) {
        try {
            File file = File.createTempFile("temporaryFile", ".doc");
            Example example = new Example(file);
            System.out.println(example.getName());
        } catch (FileSystemException e) {
            System.out.println("exception " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Another catch block for a different exception");
            System.out.println("What happened? " + e.getMessage());
        } finally {
            System.out.println("finally block");
        }
    }
}
```

try { . . . } catch { . . . }

- In the **`try`** block, we place the code that might raise an exception
- When an exception occurs, it is caught by the **`catch`** block.
- If no exception occurs, the **`catch`** block is skipped.
- There may be multiple **`catch`** blocks, in case of multiple exceptions that we want to process differently.

1) access method from File class to pass two variables in and store in a variable

2) Create an instance of Example class and pass in variable of File type

```
try {
    File file = File.createTempFile("temporaryFile", ".doc");
    Example example = new Example(file);
    System.out.println(example.getName());
} catch (FileSystemException e) {
    System.out.println("exception " + e.getMessage());
} catch (IOException e) {
    System.out.println("Another catch block for a different exception");
    System.out.println("What happened? " + e.getMessage());
}
```

2) Allowed, as Example constructor passes in one File type variable

3) Example class has a getName method

CBF ACADEMY

try { . . . } catch { . . . }

In case of **multiple `catch` clauses**, Java will match the exception thrown to the first exception type it finds, or to any of its parents.

In the example below, if the try block throws an exception which is not a `FileSystemException`, but a subclass of the `IOException`, then the second catch block will be executed.

```
try {
    File file = File.createTempFile("temporaryFile", ".doc");
    Example example = new Example(file);
    System.out.println(example.getName());
} catch (FileSystemException e) {
    System.out.println("exception " + e.getMessage());
} catch (IOException e) {
    System.out.println("Another catch block for a different exception");
    System.out.println("What happened? " + e.getMessage());
}
```


CBF ACADEMY

finally { . . . }

- The **`finally`** block is optional, but is **always executed** if present - whether an exception occurs or not.
- There can only be one **`finally`** block for each **`try`** block.
- The finally block is mostly used to clean up, like closing a connection for example.

```
try {
    File file = File.createTempFile("temporaryFile", ".doc");
    Example example = new Example(file);
    System.out.println(example.getName());
} catch (IllegalArgumentException | IOException e) {
    System.out.println("exception " + e.getMessage());
}  finally {
    System.out.println("The final block is always executed");
}
```

throws { . . . throw . . . }

- The **`throws`** keyword is used to indicate that a method may raise an exception, and declare the type of that exception
- The **`throw`** keyword is used to explicitly raise an exception

```
public String getName(File file) throws IllegalArgumentException {
    if (file == null) {
        throw new IllegalArgumentException("No file");
    }
    return file.getName();
}
```


CBF ACADEMY

Java try-with-resources

- A **`try-with-resources`** construct is a **`try-catch`** block with one or more resources. The resources are automatically closed at the end of the block.
- The resource is an object to be closed, it must be declared and initialised in the **`try`** statement, and it must implement the **`AutoClosable`** interface.

```
Path path = Paths.get("user/output/file.txt");
try (FileReader fr = new FileReader(path.toFile());
     BufferedReader br = new BufferedReader(fr)) {

    String line = br.readLine();
    System.out.println(line);

} catch (IOException e) {
    System.out.println("exception: " + e.getMessage());
}
```


CBF ACADEMY

Practice

Goal

Familiarise ourselves with handling:

- checked exceptions
- unchecked exceptions

Exercise

Complete exercises on:

- Exceptions https://github.com/cbfacademy/intro-to-java-course/tree/main/exercises/java-exceptions#pushpin-exceptions

How are you feeling?

#intro-to-java-channel

🤯

I have no idea what you’re talking about

✋

I have some questions but feel like I understand some things

👍

I feel comfortable with everything you’ve said

CBF ACADEMY

Custom Exceptions

CBF ACADEMY

Custom Exceptions

Custom exceptions are the ones we define ourselves.

Suppose we're building an accounting system, to help customers manage their funds:

- allow customers to deposit funds onto their account
- allow customers to withdraw funds from their account

We could define an exception for cases where customers attempt to withdraw more funds than currently available.

CBF ACADEMY

Custom Exceptions

Program.java

```java
package com.codingblackfemales.academy;

import com.codingblackfemales.academy.exceptions.Account;
import com.codingblackfemales.academy.exceptions.InsufficientFundsException;

public class Program {
    public static void main(String... args) {
        Account account = new Account(50.00);
        try {
            account.withdraw(100.00);
        } catch (InsufficientFundsException e) {
            System.out.println("Oops! " + e.getMessage());
        }

    }
}
```

Account.java

```java
public class Account {

    private Double balance;

    public Account(Double amount) {
        this.balance = amount;
    }

    public void withdraw(Double amount) throws InsufficientFundsException {
        if (amount > this.balance) {
            throw new InsufficientFundsException("Illegal withdrawal attempt");
        }

        this.balance = this.balance - amount;
    }

    public void deposit(Double amount) {
        this.balance = this.balance + amount;
    }
}
```

InsufficientFundsException.java

```java
package com.codingblackfemales.academy.exceptions;

public class InsufficientFundsException extends Exception {
    private final String message;

    public InsufficientFundsException(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return String.join(" - ", super.getMessage(), this.message);
    }
}
```

Custom Exceptions

A **custom** exception is a class like any other that we define ourselves.

We define custom exceptions because we want to “force” the system/developer that uses ours to handle them. Therefore, to ensure that they are handled at compile-time, they must be **checked** exceptions.

InsufficientFundsException.java

```java
package com.codingblackfemales.academy.exceptions;

public class InsufficientFundsException extends Exception {
    private final String message;

    public InsufficientFundsException(String message) {
        this.message = message;
    }

    @Override
    public String getMessage() {
        return String.join(" - ", super.getMessage(), this.message);
    }
}
```


CBF ACADEMY

Custom Exceptions - Good Practices

When defining our own exceptions, a few things to be cautious about:

- an exception should be used only in exceptional conditions - not as a result of normal control flow
- an exception should be used to indicate and address recovery in the presence of a fault in the program
- we use them very scarcely, because they are performance expensive when they happen

Practice

Goal

Familiarise ourselves with creating:

- custom checked exceptions
- custom unchecked exceptions

Exercise

Complete exercises on:

- Custom exceptions https://github.com/cbfacademy/intro-to-java-course/tree/main/exercises/java-exceptions#pushpin-custom-exceptions

How are you feeling?

#intro-to-java-channel

🤯

I have no idea what you're talking about

✋

I have some questions but feel like I understand some things

👍

I feel comfortable with everything you've said

CBF ACADEMY

Questions?

Knowledge Check