ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

OBJECT-ORIENTED PROGRAMMING

EXCEPTIONS

Autores:

- Adonny Mateo Calero Arguello
- Joan Oswaldo Cobeña Zambrano
- Jordan Alexander Guaman Alcivar
- Jhordy Paul Marcillo Mora
- Lenin David Moran Palomo
- Edison Damian Verdesoto Segovia

EXCEPTIONS IN JAVA

1922 ECUADOR
ESPE
UNIVERSIDAD DE LAS FUERZAS ARMADAS
INNOVACIÓN PARA LA EXCELENCIA
100 años

DEFINITION

Exceptions in Java are unexpected events or conditions that occur during the execution of a program and interrupt its normal flow. They represent error or exceptional situations that must be detected, controlled and handled appropriately to avoid abrupt termination of the program and allow a controlled response to such events.

EXCEPTION HANDLING

Exception handling allows you to control how errors and exceptional situations are responded to, preventing the program from stopping abruptly. Additionally, it provides the ability to take specific actions based on the type of exception, such as displaying error messages, performing recovery actions, or notifying the user.

Exception

EXAMPLES

1.- NullPointerException: Thrown when trying to access or use an object reference that has not been initialized, that is, when trying to access an object that has the value null.

```
String text = null;
int length = text.length(); // Lance NullPointerException
```

2.- ArrayIndexOutOfBoundsException: Thrown when trying to access an invalid index on an array. The index is outside the allowed range for the size of the array.

```
int[] numbers = { 1, 2, 3 };
int value = numbers[3]; // Lance ArrayIndexOutOfBoundsException
```

3.- FileNotFoundException: Thrown when trying to access a file that does not exist or cannot be opened.

```
File file = new File(pathname: "archivo.txt");
Scanner scanner = new Scanner(source: file); // Lance FileNotFoundException
```

These are just a few examples of exceptions in Java. There are many more exceptions defined in the standard Java library, and it is also possible to create custom exceptions based on the needs of the program.

<!--Estudio Shonos-->

Exceptions;

<By=" Team Number 2"/>

}

Exceptions {

An exception is an abnormal situation that can occur
when we run a certain program. The way the programmer
treat it is what is generally known as management or management of the exception.
an exception is an abnormal situation that can occur
when we run a certain program. The way the programmer
treat it is what is generally known as management or management of the exception.

Some cases of anomalous situations that can be cited are:

- call a method on a "null" object.
- Try to divide a
- number by “0”.
- try to open a file that does not exist to read it.

}

Exception´s Constructors{

`Exception()`

First parameterless constructor

`Exception (String message)`

A second constructor, with a parameter of “String” type, which allows us to create the exception with a specific message.

The first thing that should be clear is that in Java, all exceptions that we can use or create our own, they must inherit from the "Exception" class of the java library

The use of exceptions in Java allows for error handling and control in a structured way, preventing the program from stopping abruptly and providing a form of proper error handling or recovery

}

Example of exception{

```java
package exception.example;

/**
 *
 * @author Adonny Calero,Jsons,DCCO-ESPE
 */
public class Exception{
    public static void main(String[] args) {
        try {
            int dividend = 10;
            int divider = 0;
            int result = dividend / divider; // We try to divide by zero, which throws an exception
            System.out.println("The result of the division is: " + result); // This line will not be executed
        } catch (ArithmeticException e) {
            System.out.println("An arithmetic error occurred: " + e.getMessage()); // We catch and display the exception message
        }
    }
}

```

}

About the example {

In the example an attempt is made to divide by zero, which is an invalid operation. This situation generates an exception of type “ArithmeticException”. The line of code "int result = dividend / divisor;" is where the exception is thrown.

- To handle this exception, we use a try-catch block. The try block contains the code that might throw an exception. In this case, the split operation is inside the try block. If an exception occurs inside the try block, it jumps to the corresponding catch block

In the catch block, you specify the type of exception you want to catch. In this example, we use catch(ArithmeticException e), which means that we are catching exceptions of type ArithmeticException. Inside the catch block, we can perform actions to handle the exception, such as displaying an error message

- In the example, trying to divide by zero throws an ArithmeticException exception, which is caught by the catch block. Inside the catch block, the message "An arithmetic error occurred: / by zero" is returned, using the getMessage() method of the exception object (e) to get the specific message of the exception.

}

<!--Estudio Shonos-->

Thanks {

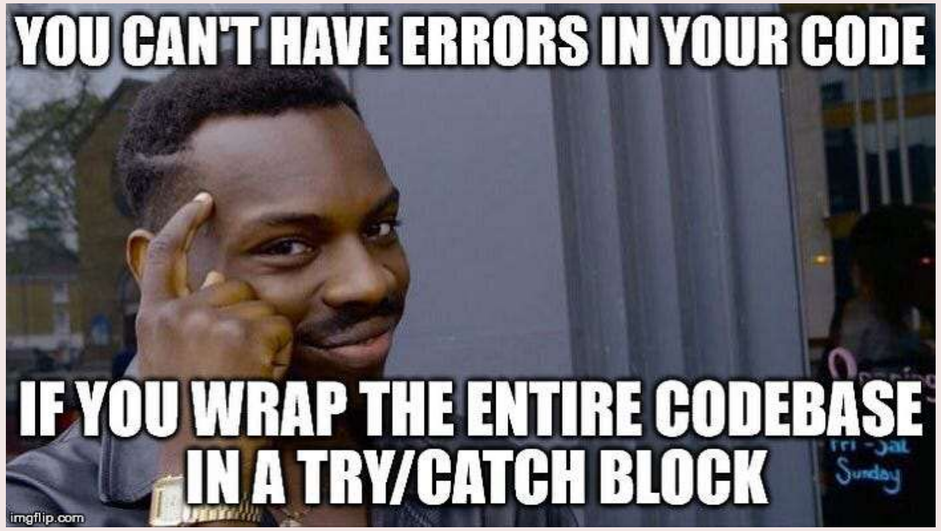
<By="Team number 2"/>

}

Exceptions

Errors

Errors



- there are many types of errors that can be used exceptions for example hardware problems. bad hard drive or bad data.
- when occurred this type of exceptions Java create a method called Throwable.
- this method have an information about the exception include the type, state for program and the moment than occurred the error

Advantaje of controller errors

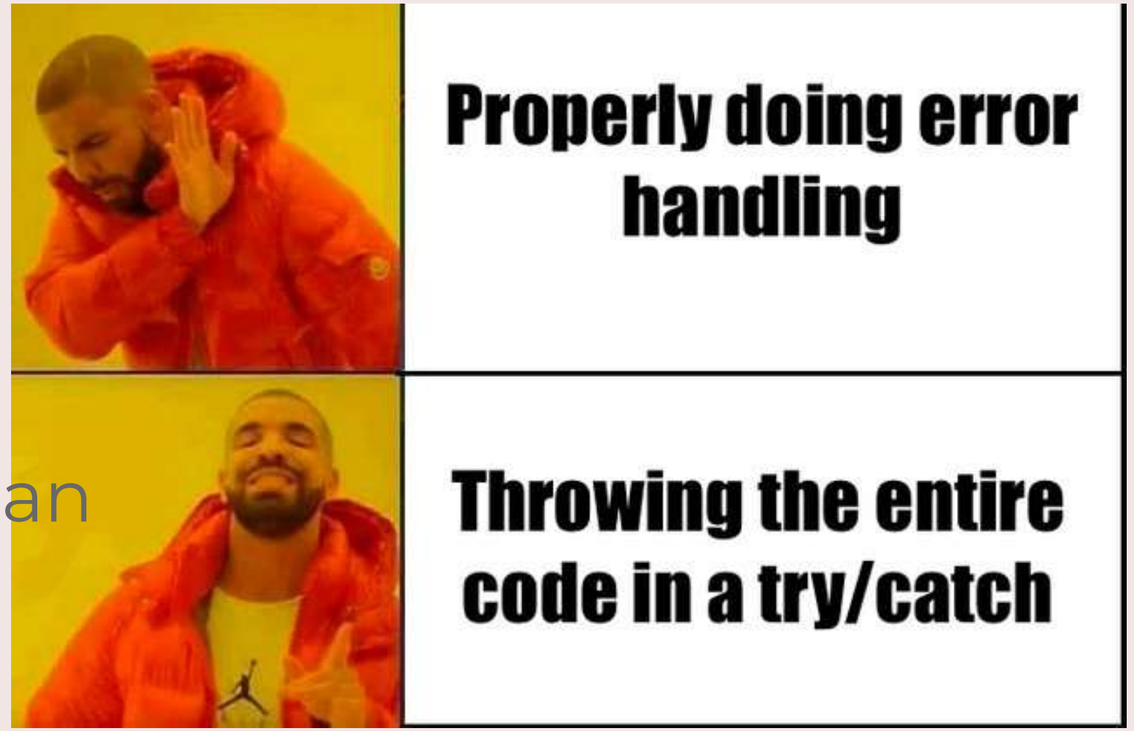
Separation of error handling from "normal" code.

Propagation of errors throughout the call stack.

Grouping of error types and differentiation between them.

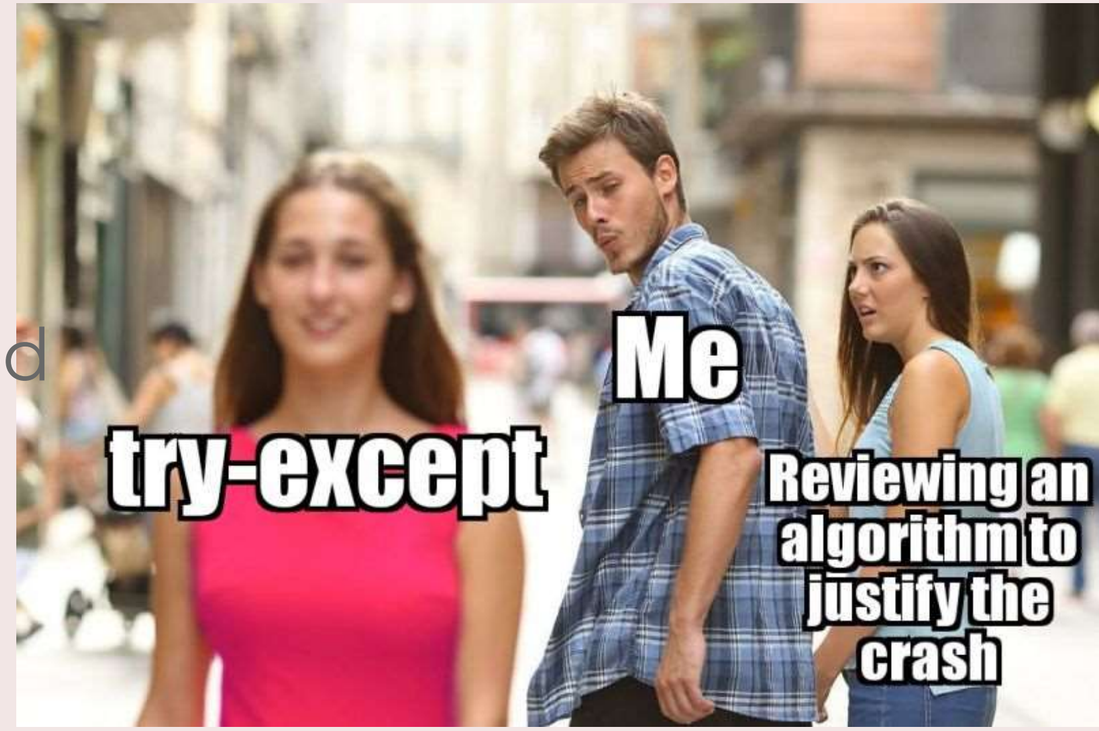
Separation of error handling from "normal" code.

- By using this method of exceptions. the mistakes have identified and controller for separated, this has a result a clean code and readable



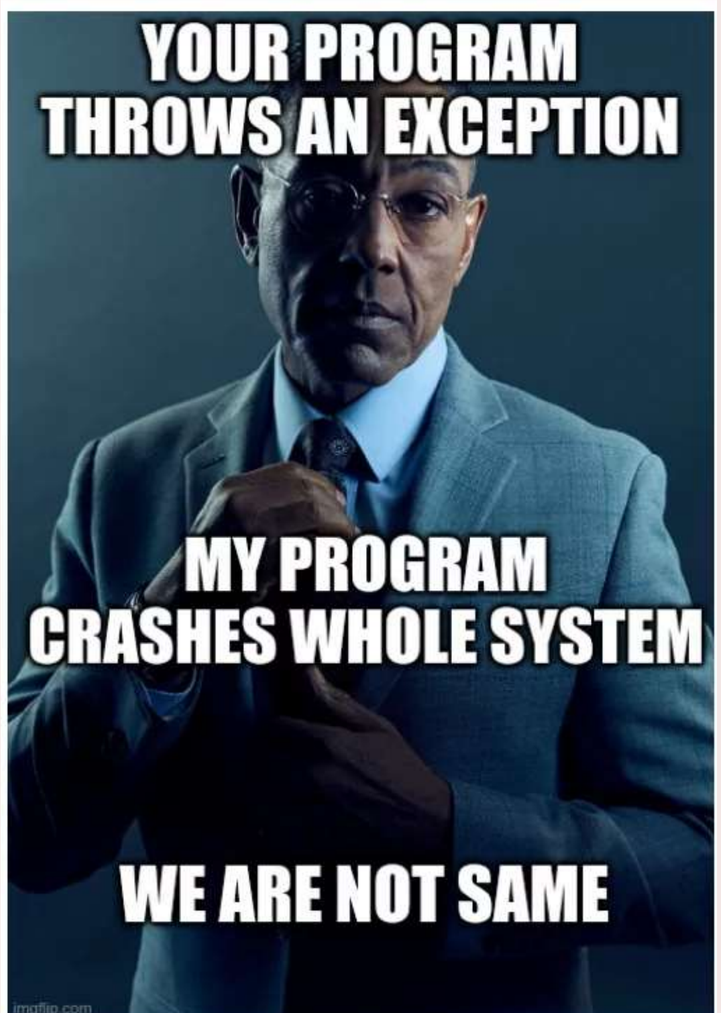
Propagation of errors throughout the call stack.

- Another advantage of the controller exceptions in Java is the capacity of spread mistakes through stack calls.
- when an exceptions occur is a method can be cast and spread a up if stack called until a find a controller of exceptions appropriate



Grouping of error types and differentiation between them.

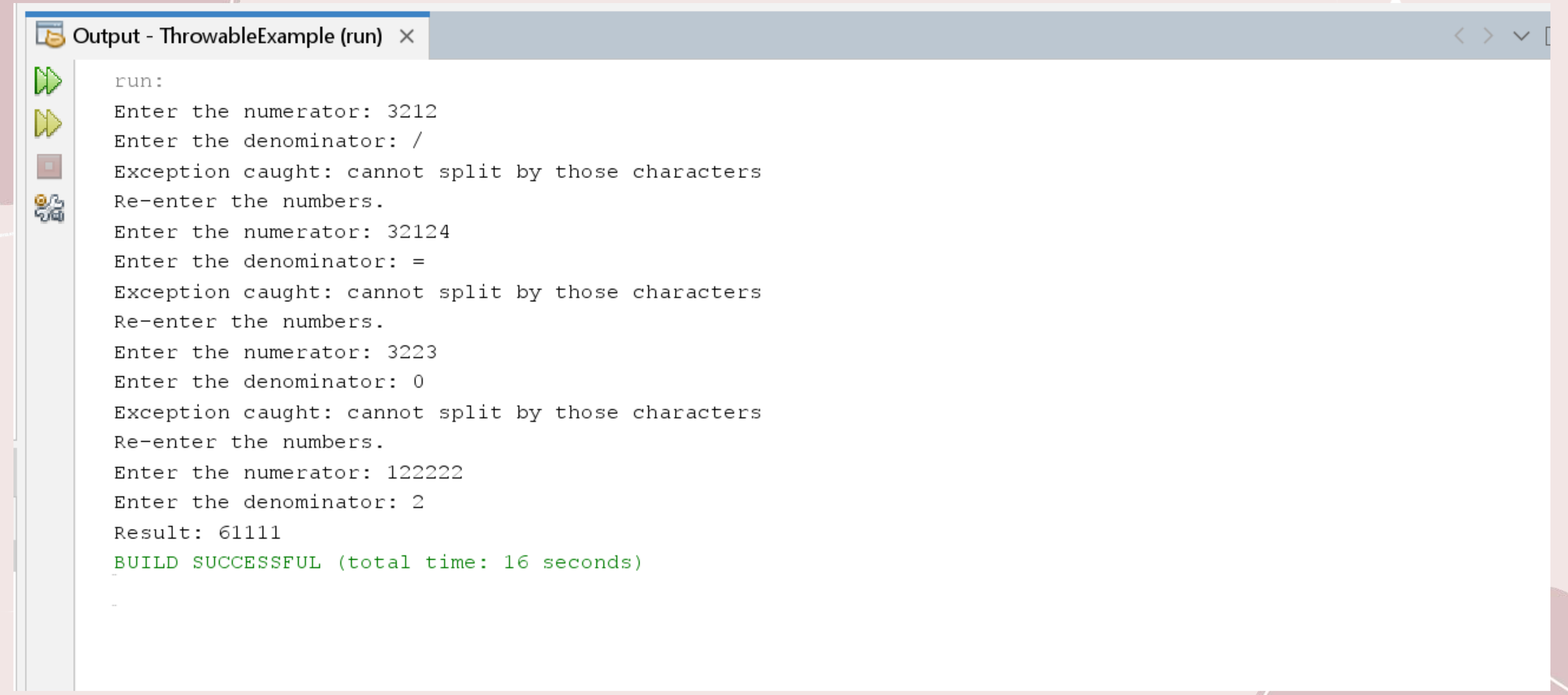
- In Java, the exceptions organized in a hierarchy of class. this it means we can a defined and catch exceptions specifics for different types of mistakes
- For example, can catch exceptions specific for mistakes of entry and exit, mistakes of connection or calculate, or more



For example

```
5   public class ThrowableExample {
6
7       public static void main(String[] args) {
8           boolean continuar = true;
9
10          while (continuar) {
11              try {
12                  Scanner scanner = new Scanner(source: System.in);
13
14                  System.out.print(s: "Enter the numerator: ");
15                  int numerator = scanner.nextInt();
16
17                  System.out.print(s: "Enter the denominator: ");
18                  int denominator = scanner.nextInt();
19
20                  divideNumbers(numerator, denominator);
21                  continuar = false;
22              } catch (Throwable t) {
23                  System.out.println(x: "Exception caught: cannot split by those characters" );
24                  System.out.println(x: "Re-enter the numbers.");
25              }
26          }
27      }
28
29      public static void divideNumbers(int numerator, int denominator) {
30          int result = numerator / denominator;
31          System.out.println("Result: " + result);
32      }
33  }
34
```

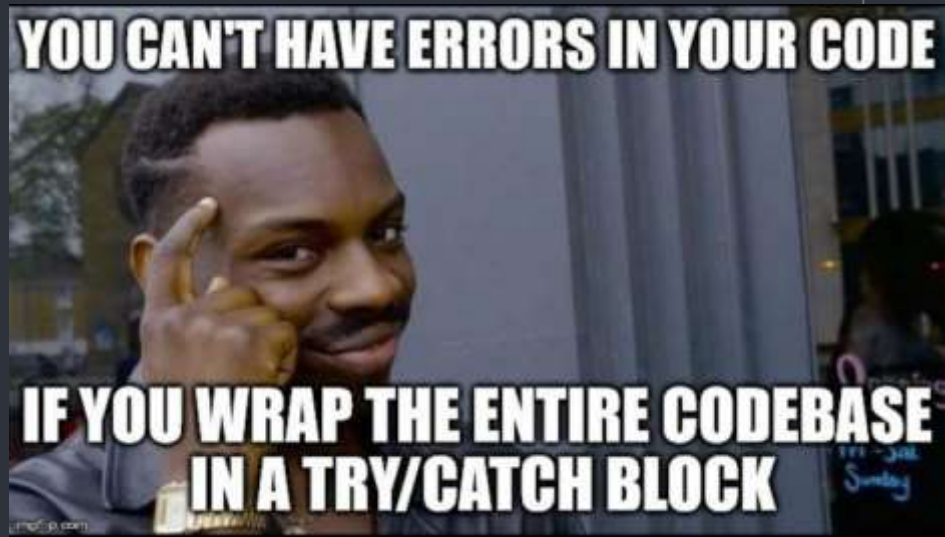
For example


Output - ThrowableExample (run)
run:
Enter the numerator: 3212
Enter the denominator: /
Exception caught: cannot split by those characters
Re-enter the numbers.
Enter the numerator: 32124
Enter the denominator: =
Exception caught: cannot split by those characters
Re-enter the numbers.
Enter the numerator: 3223
Enter the denominator: 0
Exception caught: cannot split by those characters
Re-enter the numbers.
Enter the numerator: 122222
Enter the denominator: 2
Result: 61111
BUILD SUCCESSFUL (total time: 16 seconds)

exception.java

exception.class

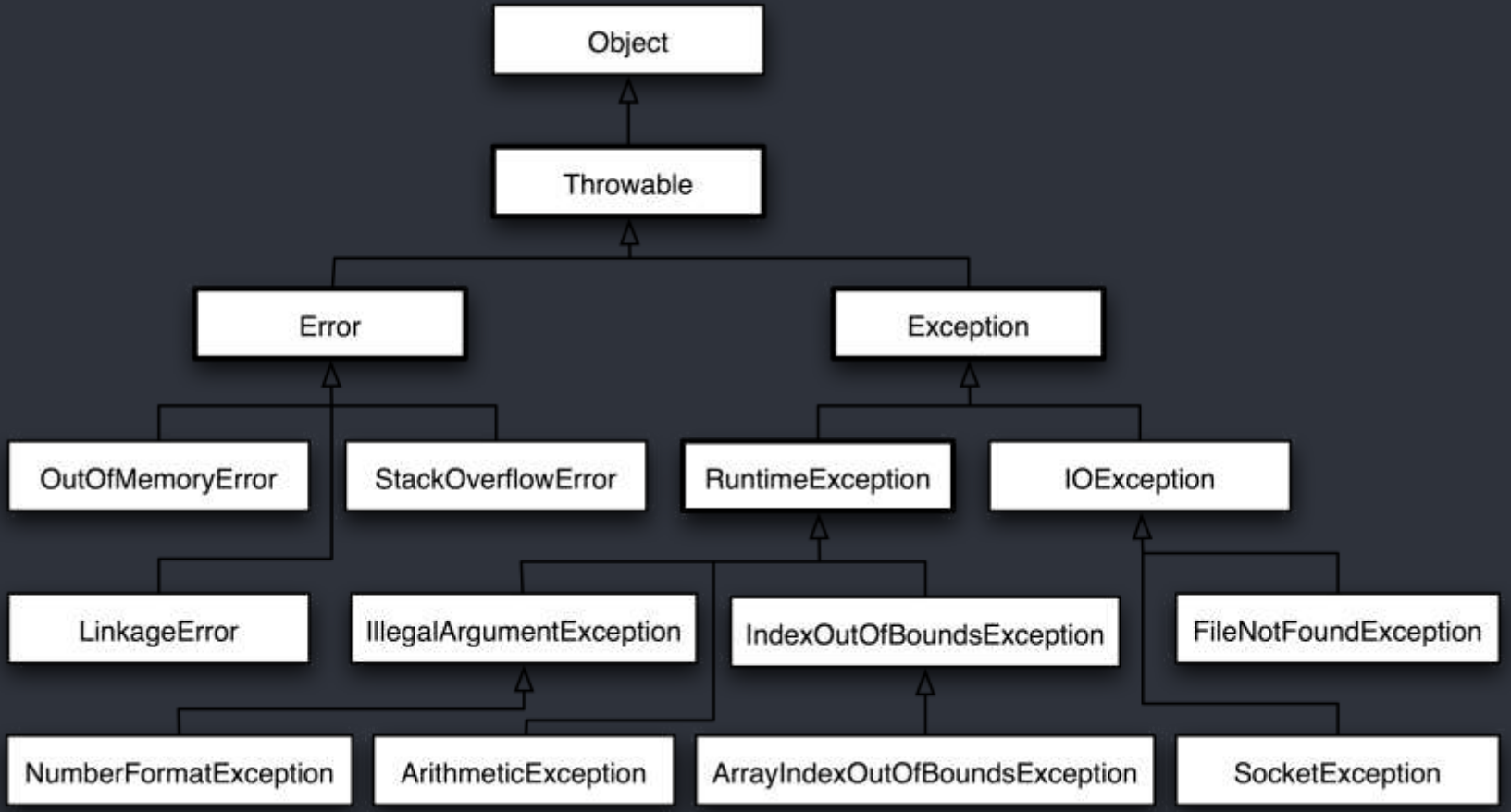
```
1   Exception.Classes {
2
3           int groupId = 2;
4
5           public static void tryToExplain() {
6                   randomMember.explain();
7           }
8   }
9
10
11
12
13
14
```



exception.java

exception.class

```
1
2     String definition = “In Java, most of the things” +
3             “use classes, even the exceptions, for” +
4             “handling they the Exception class is the base” +
5             “class for all exceptions. The Exception class” +
6             “has a set of sub-classes for handling” +
7             “different types of exceptions.” ;
8
9     //Exception class is a throwable class
10
11
12
13
14    }
```


Object
Throwable
Error
Exception
OutOfMemoryError
StackOverflowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException

Subclasses of Exception class

/01 AclNotFoundException

< This exception is thrown when a reference is made to an ACL (Access Control List) that does not exist.

/02 ActivationException

< This exception is thrown when activation fails during object deserialization.

/03 AlreadyBoundException

< This exception is thrown when an attempt is made to bind an object in the registry to a name that already has an associated binding.

Subclasses of Exception class

/04 BadAttributeValueExpException

< This exception is thrown when the value of an attribute in the directory cannot be interpreted by the directory.

/05 BadLocationException

< This exception is thrown when a bad location is accessed in a document.

/06 CertificateException

< This exception indicates one of a variety of certificate problems.

Subclasses of Exception class

/07 IOException

< Signals that an Input/Output exception of some sort has occurred.

/08 IllegalClassFormatException

< This exception indicates that there was an error while reading or parsing a class file.

/09 RuntimeException

< Is a type of exception that occurs at the time of execution.

/10 IndexOutOfBoundsException

< It is thrown when an array or vector has been accessed with an illegal index.

Types of Exceptions in Java

Exceptions are a vital part of Java programming. Let's explore the different types of exceptions and learn how to handle them effectively.

Checked Exceptions

1 What are they?

Checked exceptions are standard exceptions that are checked at compile time, requiring you to declare them in your source code.

2 Examples

FileNotFoundException, SQLException, IOException

3 How to Handle Them

You need to handle checked exceptions using a try-catch block or by declaring them with the 'throws' keyword.

Example:

```java
/*
 * @author Edison Verdesoto, Code Warriors, DCCO-ESPE
 */
public class CheckedException {
    public static void main(String[] args) {
     Scanner scanner = new Scanner(source:System.in);
       String option;
       do {
           try {
               System.out.print(s: "Enter age: ");
               int age = scanner.nextInt();
               scanner.nextLine();
               validateAge(age);
               System.out.println(x: "Valid age!");
           } catch (InvalidAgeException e) {
               System.out.println("Invalid age: " + e.getMessage());
           }

           System.out.print(s: "Do you want to enter another age? (y/n): ");
           option = scanner.nextLine();
       } while (option.equalsIgnoreCase(anotherString:"y"));

       scanner.close();
   }

   public static void validateAge(int age) throws InvalidAgeException {
       if (age < 18) {
           throw new InvalidAgeException(message: "Age must be 18 or above.");
       }
   }
}

class InvalidAgeException extends Exception {
   public InvalidAgeException(String message) {
       super(message);
   }
}
```

Unchecked Exceptions

What are they?

Unchecked exceptions are runtime exceptions that are not checked at compile time. They can occur anywhere in your program.

How to Handle Them

Since unchecked exceptions are not checked at compile time, you can handle them using a try-catch block or let them propagate up the call stack.

Examples

NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

Example:

```java
/*
 * @author Edison Verdesoto, Code Warriors, DCCO-ESPE
 */
public class CustomExceptions {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(source:System.in);
        String option;
        do {
            try {
                System.out.print(s: "Enter dividend: ");
                int dividend = scanner.nextInt();
                System.out.print(s: "Enter divisor: ");
                int divisor = scanner.nextInt();
                scanner.nextLine();
                int result = performDivision(dividend, divisor);
                System.out.println("Result: " + result);
            } catch (ArithmeticException e) {
                System.out.println(x: "Error: Division by zero is not allowed.");
            }

            System.out.print(s: "Do you want to perform another division? (y/n): ");
            option = scanner.nextLine();
        } while (option.equalsIgnoreCase(anotherString: "y"));

        scanner.close();
    }

    public static int performDivision(int dividend, int divisor) {
        return dividend / divisor;
    }
}
```

Throw vs. Throws

Throw

Throw is used to throw a new exception explicitly in your code.

Throws

Throws can be used to declare the exceptions a method can throw. This is useful when a method does not handle a particular exception, but wants to leave it to the calling method.

Final vs. Finally vs Finalize

1

Final

Final is a keyword used to modify the value of variables, prevent method overriding, or prevent class inheritance.

2

Finally

Finally is a block of code that gets executed irrespective of whether an exception is thrown or not.

3

Finalize

Finalize() is a method that's called by the garbage collector when an object is about to be destroyed. It can be overridden to perform some cleanup tasks.

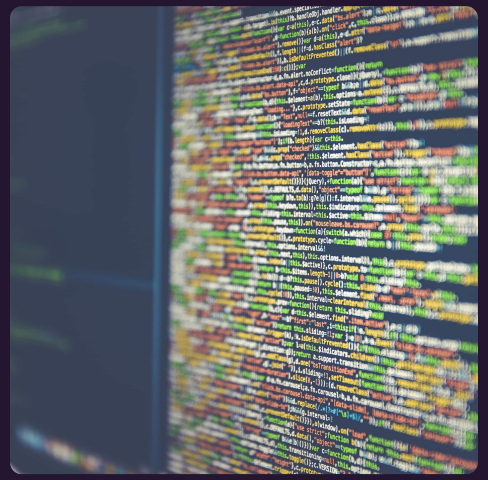
Exception Handling Best Practices

Small, Specific Try-Catch Blocks

Try to keep your try-catch blocks small and specific to the code that can potentially throw an exception.

Log Your Exceptions

Logging your exceptions can help you pinpoint the root cause of the exception and find quick solutions.



Test Your Code for Exceptions

Testing your code for exceptions can help you identify and fix problems before they get to production.

Common Exception Handling Mistakes

Ignoring Exceptions

Ignoring exceptions can cause your code to fail silently, making it difficult to find and debug issues.

Catching General Exceptions

Catching general exceptions like Exception or Throwable can catch all exceptions, including unchecked exceptions, and make it harder to pinpoint the root cause of the problem.

Not Wrapping Exceptions

Not wrapping your exceptions in a custom exception class can make it difficult to understand the cause of the exception and lead to unclear solutions.

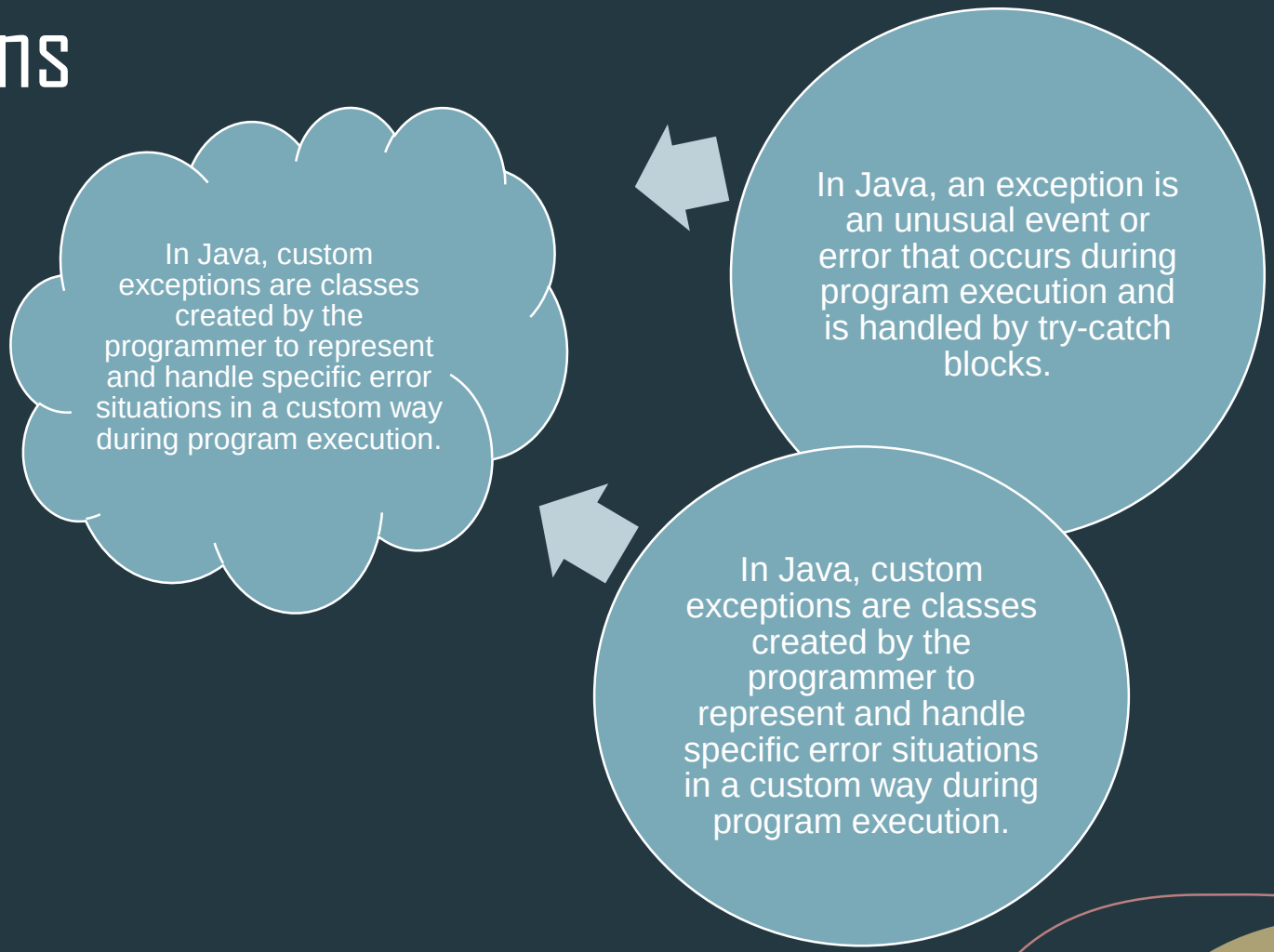
Custom Exceptions

Members:

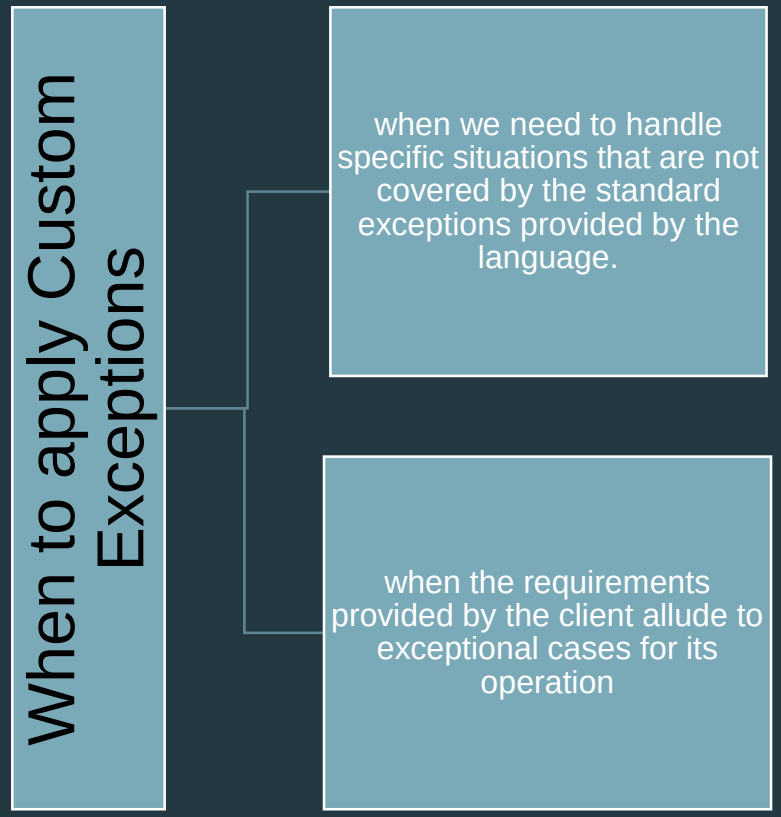
- **Calero Mateo**
- **Cobeña Joan**
- **Guaman Jordan**
- **Marcillo Jhordy**
- **Moran David**
- **Verdesoto Edison**

Proffesor: Edison Lascano

Difference Between Common Exceptions and Custom Exceptions



Application of Exceptions



Why use custom exceptions?

- By creating our own exceptions, we can define the exact behavior that we want to occur when a particular error occurs. This helps improve code readability and maintainability as developers can quickly identify what type of error has occurred and how to handle it. In addition, custom exceptions also allow us to add additional information about the error, such as specific error messages or relevant data to make debugging easier.

Steps to implement Custom Exceptions in Java

- Create a new class that extends the Exception class.
- Add constructors to your exception class to allow different ways to initialize it.
- Use your custom exception in the code, coupling the implementation syntax to the main code.

Custom Exceptions Conclusion

- Custom exceptions are a valuable technique that allows us to handle exceptional situations in a more precise and controlled way.
- By creating our own exceptions, we can improve code clarity, provide detailed error information, encourage code reuse, and make debugging easier.

Exceptions:

Exceptions are the means offered by some programming languages to deal with anomalous situations that can happen when we execute a program. Some cases of anomalous situations that can be cited are, for example, calling a method on a null object, attempting to split a number by "0", try to open a file that does not exist to read it and others.

In general, when an exception appears inside a method, it is because the method itself has created and thrown it, or because said exception has been thrown from some other method that has been called from it.

```
/**
 * @author Edison Verdesoto, Code Warriors, DCCO-ESPE
 */
public class CheckedException {
    public static void main(String[] args) {
     Scanner scanner = new Scanner(source:System.in);
       String option;
       do {
           try {
               System.out.print(s: "Enter age: ");
               int age = scanner.nextInt();
               scanner.nextLine();
               validateAge(age);
               System.out.println(x: "Valid age!");
           } catch (InvalidAgeException e) {
               System.out.println("Invalid age: " + e.getMessage());
           }

           System.out.print(s: "Do you want to enter another age? (y/n): ");
           option = scanner.nextLine();
       } while (option.equalsIgnoreCase(anotherString:"y"));

       scanner.close();
    }

    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException(message:"Age must be 18 or above.");
        }
    }
}

class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

In Java, exceptions are events that occur during the execution of a program that interrupt the normal flow of execution. When an exception occurs, an object of a specific class is thrown that represents the type of exception that occurred. These objects are called "exceptions" and contain information about the error that has occurred.

```java
/*
 * @author Edison Verdesoto, Code Warriors, DCCO-ESPE
 */
public class CustomExceptions {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(source: System.in);
        String option;
        do {
            try {
                System.out.print(s: "Enter dividend: ");
                int dividend = scanner.nextInt();
                System.out.print(s: "Enter divisor: ");
                int divisor = scanner.nextInt();
                scanner.nextLine();
                int result = performDivision(dividend, divisor);
                System.out.println("Result: " + result);
            } catch (ArithmeticException e) {
                System.out.println(x: "Error: Division by zero is not allowed.");
            }

            System.out.print(s: "Do you want to perform another division? (y/n): ");
            option = scanner.nextLine();
        } while (option.equalsIgnoreCase(anotherString: "y"));

        scanner.close();
    }

    public static int performDivision(int dividend, int divisor) {
        return dividend / divisor;
    }
}
```

Errors:

There are several types of errors that can use exceptions, from serious hardware problems, hard drive failure, or simply bad data. When such an error occurs within a Java method, the method creates an object called 'exception'. This object contains information about the exception, including its type and the state of the program at the time the error occurred.

Once a method throws an exception, the runtime system kicks in to look for an exception handler. An exception handler is considered adequate if the type of exception thrown is the same as the type of exception handled by the handler.

If the runtime system searches all the calling methods without finding the appropriate exception handler, the runtime system terminates (and therefore the Java program also terminates). By using exceptions to handle errors, Java programs have several advantages over traditional error handling techniques.

Advantage 1: Separation of error handling from "normal" code.

By using exceptions, errors can be identified and handled separately, resulting in cleaner, more readable code. Instead of having error handling logic scattered throughout the main code, "try-catch" blocks can be used to centrally catch and handle exceptions. This makes the code easier to understand and maintain.

Advantage 2: Propagation of errors throughout the call stack.

Another important advantage of exception handling in Java is the ability to propagate errors throughout the call stack. When an exception occurs in a method, it can be thrown and propagated up the call stack until a suitable exception handler is found that can handle it.

This allows the error to be caught and dealt with at the appropriate level of the method call hierarchy.

Advantage 3: Grouping of error types and differentiation between them.

In Java, exceptions are organized in a hierarchy of classes, with more specific exceptions inheriting from more general exceptions. This means that specific exceptions can be defined and caught for different types of errors, making it easy to selectively handle each type of exceptional situation. For example, specific exceptions can be caught for input/output errors, network connection errors, or math calculation errors, among others.

Exceptions Classes

In Java, the Exception class is the base class for all exceptions. The Exception class has a set of sub-classes for handling different types of exceptions.

Those are the direct subclasses of Exception class:

- AclNotFoundException
- ActivationException
- AlreadyBoundException
- ApplicationException
- AWTException
- BackingStoreException
- BadAttributeValueExpException
- BadBinaryOpValueExpException
- BadLocationException
- BadStringOperationException
- BrokenBarrierException
- CertificateException
- CloneNotSupportedException
- DataFormatException
- DatatypeConfigurationException
- DestroyFailedException
- ExecutionException
- ExpandVetoException
- FontFormatException
- GeneralSecurityException
- GSSException
- IllegalClassFormatException
- InterruptedException
- IntrospectionException
- InvalidApplicationException
- InvalidMidiDataException
- InvalidPreferencesFormatException

- InvalidTargetObjectTypeException
- IOException

The class Exception and any subclasses that are not also subclasses of RuntimeException are checked exceptions.

Types Of Exceptions

Exceptions play a crucial role in Java programming by handling unexpected situations that can disrupt the normal execution of a program. Java encompasses various types of exceptions, each serving a unique purpose.

We classify this types as:

Checked exceptions: which necessitate explicit declaration in the method signature and require handling by the calling code. These exceptions arise from external factors beyond the program's control, such as network connectivity problems or file I/O errors. By enforcing the handling of checked exceptions, Java encourages a proactive approach to error management, compelling developers to explicitly address potential failures.
This type of exception is handled by using a try-catch block or from the “throws” keyword.

Unchecked exceptions: Unlike checked exceptions, they do not demand explicit declaration or catching in the calling code. Unchecked exceptions usually result from programming errors, like null pointer dereferences or exceeding array bounds. Since these exceptions are considered a consequence of developer mistakes, Java does not enforce explicit handling. Nevertheless, it is considered good practice to anticipate and handle unchecked exceptions appropriately to avoid unexpected program terminations.

Java also provides developers with the capability to create **custom exceptions** using user-defined exception classes. These exceptions extend either the built-in Exception class or its subclasses, such as RuntimeException. Custom exceptions enable developers to tailor their exception handling mechanisms to suit their application's specific requirements. Additionally, custom exceptions can encapsulate additional error information, simplifying the identification and resolution of issues within a program.

Custom Exceptions:
Custom exceptions are those that are created to handle specific exceptional situations **that are not covered by the standard exceptions** provided by the language.

The main difference between "standard" exceptions and custom exceptions lies in their purpose and use. Predefined exceptions are meant to cover general error cases, while custom exceptions are used when specialized error handling is needed for specific situations within an application. Custom exceptions allow greater flexibility and control in error handling, as the programmer can define his own set of exceptions and provide custom behavior for each one.

References:

GeeksforGeeks. (2022). Exceptions in Java. *GeeksforGeeks*. https://www.geeksforgeeks.org/exceptions-in-java/

Ciberaula. (s. f.). *Manejo de Errores y Excepciones en Java*. https://www.ciberaula.com/cursos/java/manejo_errores_excepciones_java.php