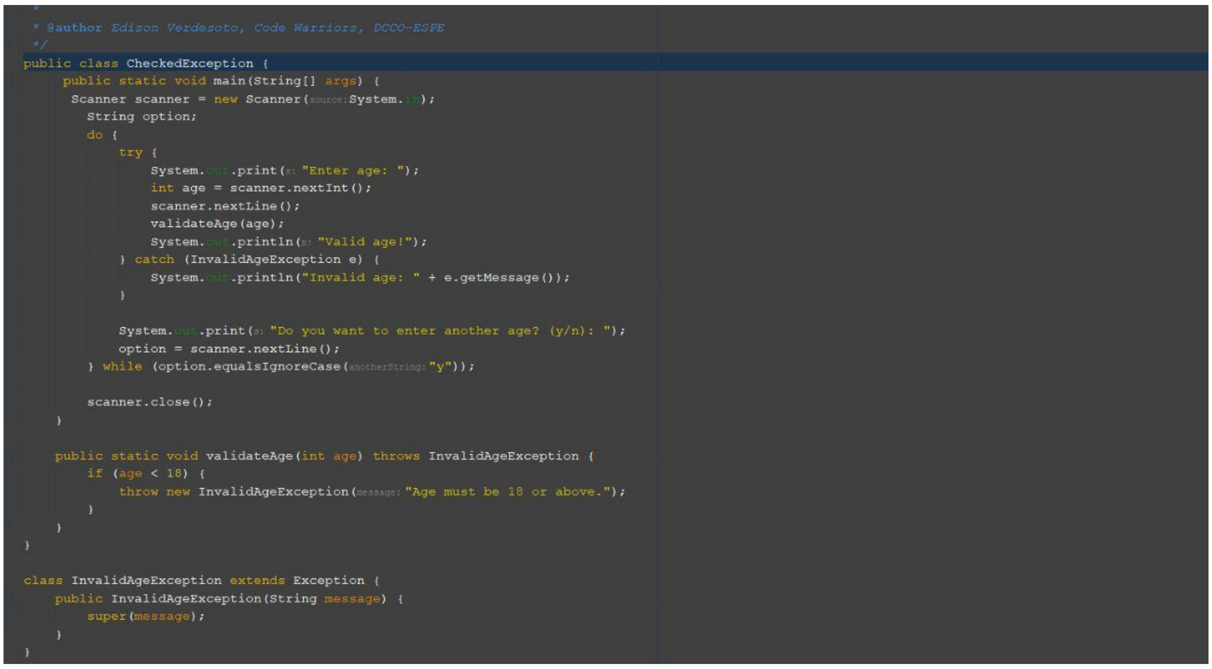
Exceptions:

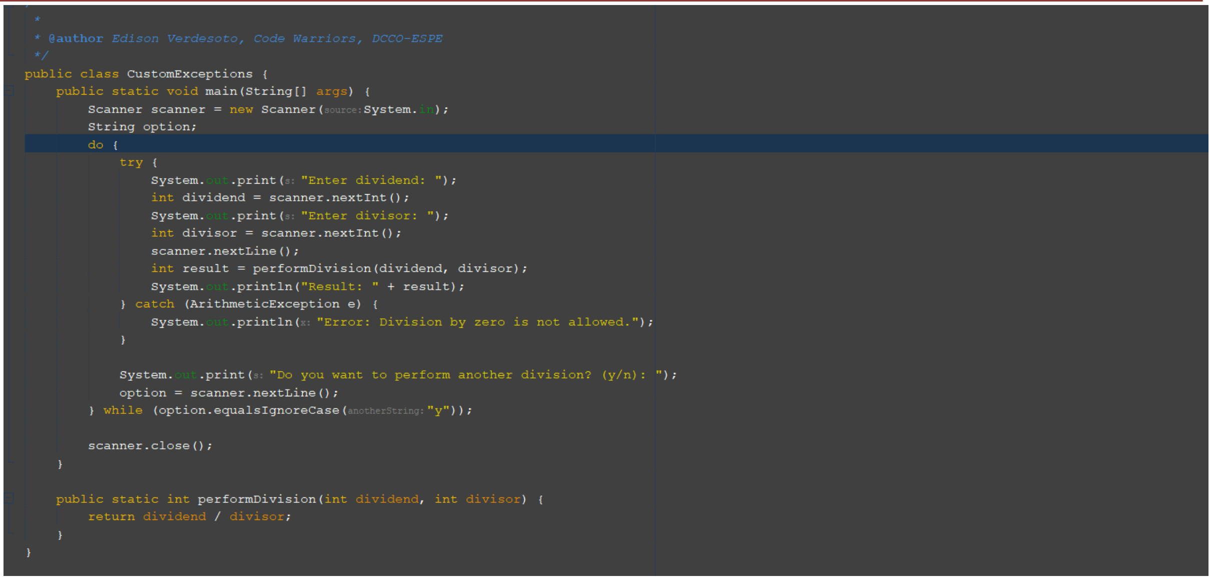
Exceptions are the means offered by some programming languages to deal with anomalous situations that can happen when we execute a program. Some cases of anomalous situations that can be cited are, for example, calling a method on a null object, attempting to split a number by "0", try to open a file 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.



```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);
    }
}
```

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.



```
 *
 * @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