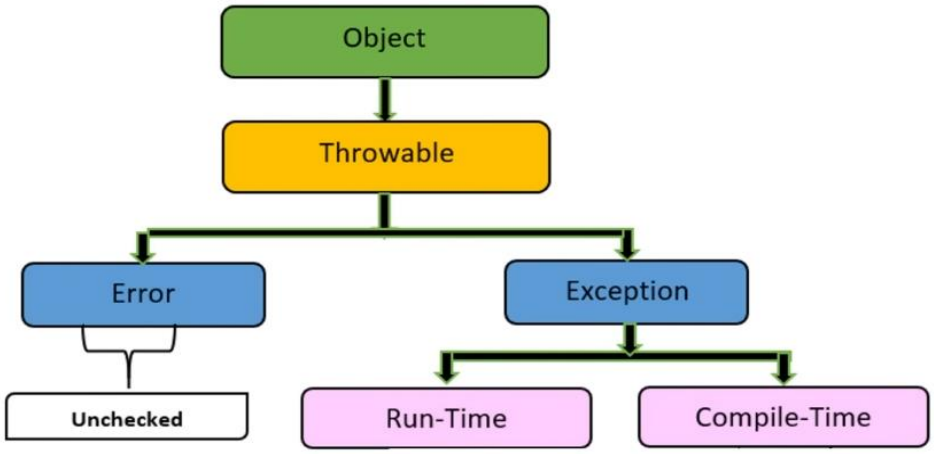
10. Exception and File Handling

10.1) What is an Exception

Exception is a disruptive event that interrupts the normal flow of execution of a program.

It's an instance of a problem that arises when program is running.

Exceptions are objects that encapsualtes the info about he error and that info includes type and state of the progarm when the error occurred.



- Errors are not under the control of the programmer.
- Errors generally represent serious problems that are outside the scope of application handling (e.g., OutOfMemoryError) and are considered unrecoverable.
- Exceptions are of two types:
  1. Runtime exceptions (unchecked)
  2. Compile-time exceptions (checked)
- Exceptions occur due to logical errors made by the programmer.
- ArrayIndexOutOfBoundsException and ArithmeticException are examples of runtime exceptions.
- Compile-time exceptions (also known as checked exceptions) are exceptions that the compiler knows may occur, and therefore must be handled in advance using try-catch blocks or by declaring them with throws.
- As shown in the diagram , the Object class is the parent of all classes that do not explicitly extend another class.
- Throwable is a class (not abstract) that extends Object and acts as the superclass for all errors and exceptions.
- Error and Exception are two direct subclasses of Throwable.

10.2) Try-Catch

Exceptions are classes.

Try block is the block of code that causes the exception.

Catch block is the block of code that catches and handles the exception thrown by the try block.

If your code has multiple catch blocks for different exceptions, only the first exception that is thrown in the try block and matches a catch block will be caught. Once an exception is caught by a catch block, no further catch blocks will be checked for that exception.

Below is how you handle a single exception.

```
package Lecture10;

import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: "); // Enter numerator: 5
        a = s.nextInt();
        System.out.print("Enter denominator:"); // Enter denominator: 0
        b = s.nextInt();
        try {
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage()); // by zero
            System.out.println("can't divide with zero"); // can't divide with zero
        }
    }
}
```

when writing multiple catch blocks in Java, the general rule is that the more specific exceptions should be caught first, followed by the more general ones.

- Most specific exceptions should come first (like ArithmeticException).
- More general exceptions like Exception should come after specific ones.
- The most general Throwable class should be the last one, as it can catch any exception or error that wasn't caught earlier.

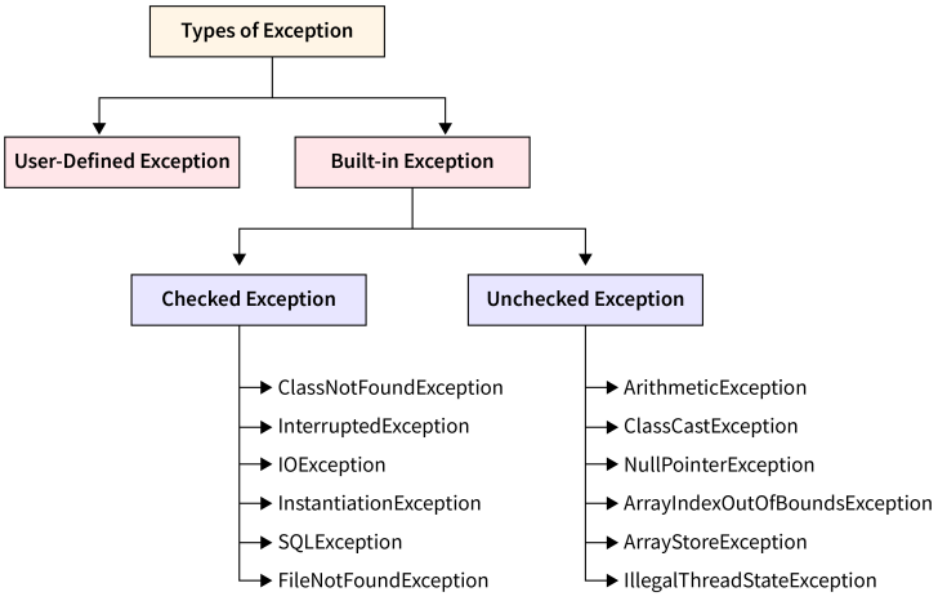
We can write multiple try-catch blocks.

```
package Lecture10;

import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: "); // Enter numerator: 5
        a = s.nextInt();
        System.out.print("Enter denominator: "); // Enter denominator: 0
        b = s.nextInt();
        try {
            // This will cause ArrayIndexOutOfBoundsException
            int[] arr = new int[5];
            arr[5] = 10; // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {
            // Catch both exceptions in a single block
            System.out.println("Exception caught: " + e.getMessage());
            if (e instanceof ArithmeticException) {
                System.out.println("Can't divide by zero.");
            } else if (e instanceof ArrayIndexOutOfBoundsException) {
                System.out.println("Limit of array exceeded.");
```

```
            }
        } catch (Exception ex) {
            System.out.println("General exception");
        } catch (Throwable t) {
            System.out.println("Throwable Exception.");
        }
        try {
            // Now perform the division operation, which could throw ArithmeticException
            System.out.println("Division: " + (a / b)); // This can throw ArithmeticException
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage());
            System.out.println("Can't divide with zero");
        } catch (Exception ex) {
            System.out.println("General exception");
        } catch (Throwable t) {
            System.out.println("Throwable Exception.");
        }
    }
}
```

10.3) Types of Exceptions



Checked exceptions are those exceptions that must be either caught or declared in the method using throws keyword.

Unchecked exceptions are need not to be handled explicitly.

All the exceptions we handles above are the unchecked exceptions.

Checked exceptions must be handled and it's your wish to handle unchecked exceptions.

User-defined exceptions in Java are custom exception classes that you create yourself to handle specific, application-related error.

10.4) Throw and throws keyword

Throws keyword

Used in method declarations to indicate that a method might throw one or more checked exceptions.

When a method declares that it throws a checked exception using throws, any method that calls it must either:

- ✓ Handle the exception using try-catch, or
- ✓ Declare it again using throws

This is similar to abstract classes: if a class inherits an abstract method, it must override it, unless it's also abstract.

```
public void methodName() throws IOException {
    // code that might throw IOException
}
```

Only used for checked exceptions (e.g., IOException, SQLException), not for unchecked ones like ArithmeticException.

```
class Calculator {

    // Method that declares an exception using throws
    public void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println("Division: " + (a / b));
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        try {
            calc.divide(10, 0);  // Must handle the exception because divide() throws
ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Throw keyword

Used to explicitly throw an exception(either new or an existing exception object) form a method or a block of code.

You can throw any Throwable (checked or unchecked).

Ex: throw new ArithmeticException("Division by zero");

exit code = 0: program ran successfully.

exit code ≠ 0: abnormal termination (usually due to an unhandled exception).

```
package Lecture10;

import java.util.Scanner;
public class Calculator {
    public static void main(String[] args) {
        a();
    }
```

```java
    public static void a() {
        b();
    }
    public static void b() {
        c();
    }
    public static void c() {
        d();
    }
    public static void d() {
        Scanner s = new Scanner(System.in);
        int a, b;
        System.out.print("Enter numerator: ");
        a = s.nextInt();
        System.out.print("Enter denominator:");
        b = s.nextInt();
        try {
            int[] arr = new int[5];
            arr[5] = 10;
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException hurrrayyy) {
            System.out.println(hurrrayyy.getMessage());
            System.out.println("can't divide with zero");
        } catch (Throwable t) {
            System.out.println("Limit of array exceeded.");
            throw t;
        }
    }
}
```

Below is the calling stack of the above progarm

```
>>
Enter numerator: 5
Enter denominator:0
Limit of array exceeded.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for
length 5
        at Lecture10.Calculator.d(Calculator.java:33)
        at Lecture10.Calculator.c(Calculator.java:20)
        at Lecture10.Calculator.b(Calculator.java:16)
        at Lecture10.Calculator.a(Calculator.java:12)
        at Lecture10.Calculator.main(Calculator.java:7)
```

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exceptions cannot be propagated using throw.	Checked exceptions can be propagated with throws.
Syntax: throw is followed by an instance of an exception class.	Syntax: throws is followed by exception class names.
Example: throw new NumberFormatException("Invalid input");	Example: throws IOException, SQLException
throw is used inside a method body.	throws is used in the method declaration (signature).
You can throw only one exception at a time using throw.	You can declare multiple exceptions using throws.
Example: throw new IOException("Connection failed!!");	Example: public void method() throws IOException, SQLException

```java
public class ThrowDemo {
    public static void readFile() throws IOException {
        throw new IOException("File not found");
    }

    public static void main(String[] args) throws IOException {
        readFile(); // exception passed to JVM
    }
}
```

Here the exception is not handled by main() methods, it also escaped by defining throws in its signature.

Here is how you create and handle a user-defined exception.

```java
// This is a checked exception because it extends Exception
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

```java
public class VoterEligibility {

    public static void checkEligibility(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above to vote.");
        }
        System.out.println("You are eligible to vote!");
    }
}
```

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your age: ");
        int age = sc.nextInt();
        try {
            VoterEligibility.checkEligibility(age);
        } catch (InvalidAgeException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

10.5) Finally block

Nested try-catch block is also possible in Java.

The finally block executes compulsorily, regardless of whether an exception occurs or not.

It is ideal for closing resources like files or database connections to prevent resource leaks.

```java
try {
    try {
        // Inner try block logic
    } catch (Exception e1) {
```

```
        // Inner catch block
    } catch (AnotherException e2) {
        // Another inner catch block
    } finally {
        // Inner finally block (always executes)
    }
} catch (OuterException e3) {
    // Outer catch block
} catch (Exception e4) {
    // Another outer catch block
} finally {
    // Outer finally block (always executes)
}
```

10.6) Custom exceptions

These are the user-defined exception classes that extend either Exception(for checked exceptions) or RuntimeException(for unchecked exceptions).

```
package Lecture10;

public class CustomException extends RuntimeException {
    private boolean ticketPurchased;
    public CustomException(boolean ticketPurchased) {
        this.ticketPurchased = ticketPurchased;
    }
    public boolean isTicketPurchased() {
        return ticketPurchased;
    }
    @Override
    public String getMessage() {
        return "you can't enter the theater";
    }
}
```

```
package Lecture10;

import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        System.out.print("Did you purchased the ticket: ");
        boolean bool = s.nextBoolean();
        try {
            if (!bool) {
                throw new CustomException(bool);
            } else {
                System.out.println("You can gooo!");
            }
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println("General exception: " + e.getMessage());
        } finally {
            System.out.println("Thaks for using this service");
        }
    }
}
```

If we enter a number instead of true/false then it will be caught by the generic catch (Exception e) block in our code.

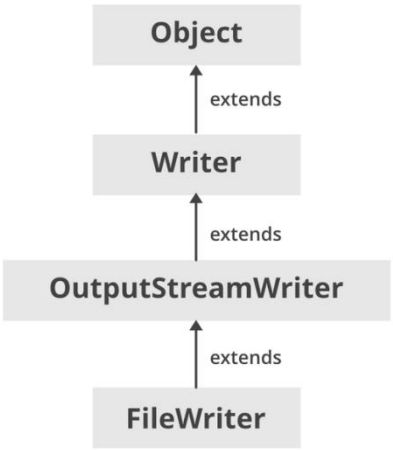
CHALLENGE

Write a program that asks the user to enter 2 integers and then divides first by second. The program should handle any arithmetic exceptions that may occur (like division by 0) and display an appropriate message.

```java
package Lecture10;

import java.util.Scanner;
public class DivisonBy0 {
    public static void main(String[] args) {
        try {
            Scanner s = new Scanner(System.in);
            int a, b;
            System.out.print("Enter 1st num: "); // true
            a = s.nextInt();
            System.out.print("Enter 2nd num: ");
            b = s.nextInt();
            System.out.println("Division: " + (a / b));
        } catch (ArithmeticException e) {
            System.out.println("Invalid input \nArithmetic exception occured");
        } catch (Exception e) {
            System.out.println("General exception: " + e.getMessage()); // General exception:
For input string: "true"
        } finally {
            System.out.println("Thank you for using this program!"); // Thank you for using
this program!
        }
    }
}
```

10.7) FileWriter class



FileWriter is used to write a stream of characters into files.

Prefer writing text over binary data.

Constructor can be created in 2 ways

- ✓ FileWriter(String filename) → creates a FileWriter obj given the name of the file to write to.
- ✓ FileWriter(File f) → creates a FileWriter obj given a file object.

flush() → forcefully clear any buffered data and send it to the intended destination I.e., file.

close() ⟶ Releases the resources.

close() automatically calls flush(), but calling flush() manually is useful when you want to write intermediate data without closing the stream.

```
import java.io.FileWriter;

public class FileWriting {
    public static void main(String[] args) {
        String fname = "sample.txt";
        try (FileWriter fw = new FileWriter(fname)) {
            fw.write("Hello world!");
            fw.flush();
            System.out.println("File written succesfully");
        } catch (Exception e) {
            System.out.println("General Exception: " + e.getMessage());
        }
    }
}
```

If we use the above try with resource method then we don't need to explicitly close the file writer.

OR

```
import java.io.FileWriter;

public class FileWriting {
    public static void main(String[] args) {
        String fname = "sample.txt";
        try {
            FileWriter fw = new FileWriter(fname);
            fw.write("Hello world!");
            fw.flush();
            fw.write("My name is Hemanth");
            fw.flush();
            fw.close();
            System.out.println("File written succesfully");
        } catch (Exception e) {
            System.out.println("General Exception: " + e.getMessage());
        }
    }
}
```

10.8) FileReader

Used to read the stream of characters from files.

It's a character based stream meaning it reads characters.

Procedure of the Constructor creation is as same as of FileWriter.

Common methods are,

read() ⟶ reads single character and returns it as an integer, returns -1 if the end of the stream is reached.

read(char[] cbuff) ⟶ reads the charcters into an array also known as cbuf(character buffer) and returns the no.of characters read.

once you've read to the end with FileReader, the only reliable way to re-read is to reopen the file.

```
import java.io.FileReader;
import java.io.IOException;
```

```java
public class FileReading {
    public static void main(String[] args) {
        String fname = "/C://Users//91868//OneDrive//Desktop//JAVA//CODE//sample.txt/";
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
            while ((val = fr.read()) != -1) {
                System.out.print((char) val); // Hello world!My name is Hemanth
            }
        } catch (IOException e) {
            System.out.println("IOException occred: " + e.getMessage());
        }
    }
}
```

CHALLENGE

File not found exception handling.

Write a program to read a filename from the user and display its content. The progarm should handle the situation where the file doesnot exists.

```java
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class Challenge {
    public static void main(String[] args) throws IOException {
        Scanner s = new Scanner(System.in);
        String fname;
        System.out.print("Enter the file name:");
        fname = s.next();
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
            while ((val = fr.read()) != -1) {
                System.out.print((char) val);
            }
        } catch (FileNotFoundException f) {
            System.out.println("file can not be found: " + f.getMessage());
        }
    }
}
```

OR

```java
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class Challenge {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        String fname;
        System.out.print("Enter the file name:");
        fname = s.next();
        int val = 0;
        try (FileReader fr = new FileReader(fname)) {
            while ((val = fr.read()) != -1) {
                System.out.print((char) val);
            }
```

```
        } catch (FileNotFoundException f) {
            System.out.println("file can not be found: " + f.getMessage());
        } catch (IOException e) {
            System.out.println("IOException occured: " + e.getMessage());
        }
    }
}
```

Below throws an error because FileNotFoundException itself is a child class of IOException.

```
catch (FileNotFoundException | IOException f) {
            if (f instanceof FileNotFoundException) {
                System.out.println("File not found : " + f.getMessage());
            } else {
                System.out.println("IOException: " + f.getMessage());
            }
        }
```

KEY POINTS

- A method that throws a checked exception must declare the exception using the throws keyword in its signature.
- We can handle more than 1 exception in a single catch block.
- Finally block will always be executed even if try-catch have return statements.
- Catch block will not be executed if an error isn't occurred in the try block.
- Unchecked exceptions are direct subclass of Exception not Throwable but Exception is direct subclass of Throwable.
- A catch block can not exist independently without try block.
- throw keyword is used to propagate an exception up the call stack.