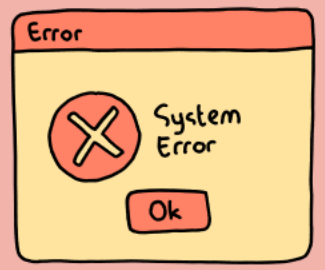
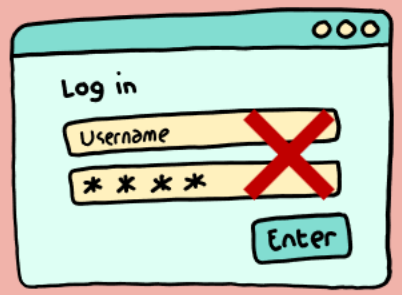
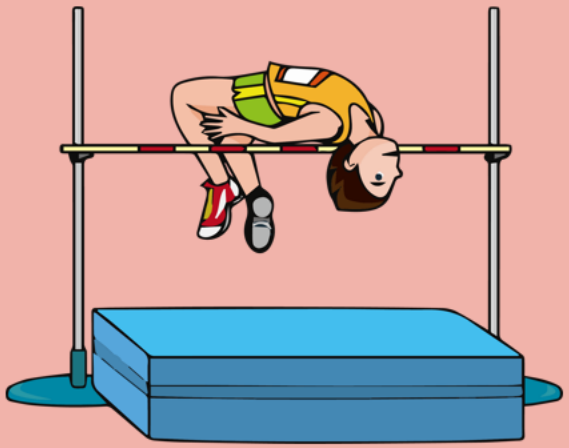
Exception handling

eazy bytes





> Irrespective of your programming proficiency, it's impossible to exercise complete control over everything. There is always a possibility of things going extremely wrong like System/Network errors, File Not Found, Invalid input, file not responding etc.
>
> When you write a risky method, you should be ready and handle the bad things that might happen. This is very similar to safety measures that we take during adventure activities.



But how do we know if a method is risky? And where do we put the code to handle the exceptional or risk situations? More details to follow.....

Taste of first Exception

eazy bytes

```java
import java.util.Scanner;

public class ExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int input = scanner.nextInt();
        System.out.println(input);
        scanner.close();
    }
}
```

This program reads a number input from the user using the Scanner class and stores it in an int variable called input. It then prints the value of input to the console.

But it does not handle any exceptions that may be thrown if the user enters invalid input. If the user enters a non-numeric value, a InputMismatchException will be thrown by the nextInt() method.

```
Enter a number: three
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at ExceptionDemo.main(ExceptionDemo.java:8)
```

Demo of ArrayIndexOutOfBoundsException

eazy bytes

```java
public class ExceptionDemo {

    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println(numbers[5]);
    }
}
```

This program creates an integer array numbers of size 5 and initializes its elements. It then tries to access the element at index 5 using numbers[5]. However, since arrays in Java are zero-indexed, the highest valid index for an array of size 5 is 4, and attempting to access an element at index 5 will result in an ArrayIndexOutOfBoundsException.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException Create breakpoint : Index 5 out of bounds for length 5
    at ExceptionDemo.main(ExceptionDemo.java:7)
```

If you run this program, it will terminate with an error message indicating that an ArrayIndexOutOfBoundsException was thrown.

try-catch block

eazy bytes

In Java, a try-catch block is used to handle exceptions that may occur during the execution of a program. An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. It is an object that represents an error or a problem that has occurred in the program.

Here's an example of a try-catch block in Java:

```
try {
    // Some code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
}
```

The try block contains the code that might throw an exception. If an exception is thrown, the catch block is executed. The catch block contains code to handle the exception. The exception type in the catch block specifies the type of exception that it can handle.

try-catch block

eazy bytes

Below are the sample solutions to handle the exceptions that we discussed previously.

```java
public class ExceptionDemo {

    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter a number....");
            int number = scanner.nextInt();
            System.out.println(number);
            scanner.close();
        } catch(Exception ex) {
            System.out.println("Please provide input in numerical format only
                  and try again...");
        }

    }

}
```

```java
public class ArrayIndexOutofBoundsExceptionDemo {

    private static Logger logger = Logger.getLogger(
            ArrayIndexOutofBoundsExceptionDemo.class.getName());

    public static void main(String[] args) {
        try {
            int[] numbers = {1,2,3,4,5};
            System.out.println(numbers[5]);
        } catch (Exception ex) {
            logger.severe("Invalid Array index. Please try again with a valid
                  index number");
        }

    }

}
```

Instead of mentioning a specific exception like InputMismatchException, we can also mention generic exception inside the catch block as shown here,

```
try{

}
catch (Exception ex) {
    // code
}
```

try-catch block

eazy bytes

You can also use multiple catch blocks to handle different types of exceptions. For example:

```
try {
  //  Some code that might throw an exception
} catch (NullPointerException ex) {
  //  Code to handle NullPointerException
} catch (StringIndexOutOfBoundsException ex) {
  //  Code to handle StringIndexOutOfBoundsException
} catch (Exception ex) {
    //  Code to handle any other exception
}
```

In this example, if a NullPointerException is thrown, the first catch block will handle it. If an StringIndexOutOfBoundsException is thrown, the second catch block will handle it. If any other type of exception is thrown, the third catch block will handle it.

When using multiple catch blocks in Java, it's important to consider the hierarchy of exception types. More specific exception types should be caught first, with more general exception types caught later. If broader exceptions are placed above smaller exceptions, it can render the more specific catch blocks useless. Therefore, the catch block with the broadest exception type should be placed at the bottom of the list of catch blocks.

Handling multiple exceptions using a single catch block

eazy bytes

```
try {
   //  Some code that might throw an exception
} catch (ArrayIndexOutOfBoundsException ex) {
   //  Code to handle ArrayIndexOutOfBoundsException
} catch (StringIndexOutOfBoundsException ex) {
   //  Code to handle StringIndexOutOfBoundsException
}
```

Before Java 7, suppose if you have the same action need to be performed in the case of different exceptions like ArrayIndexOutOfBoundsException & StringIndexOutOfBoundsException, we will forced to write multiple catch blocks like below,

To reduce the code duplication in the scenarios where we have a common action/business logic need to be performed for different exceptions we can use a single catch block with multiple exceptions separated by a | operator as shown here.

```
try {
   //  Some code that might throw an exception
} catch (ArrayIndexOutOfBoundsException | StringIndexOutOfBoundsException
                ex) {
   //  Code to handle both excptions
}
```

When you catch multiple exceptions using | , the exception variable is implicitly final. So, you cannot assign a new value to ex in the body of the catch clause

finally block

eazy bytes

In Java, a finally block is a block of code that is used to ensure that a certain block of code is always executed, regardless of whether an exception is thrown or not. Here's an example of how a finally block can be used in Java:

```
try {
    // some code that might throw an exception
} catch (ExceptionType1 e) {
    // code to handle the first type of exception
} catch (ExceptionType2 e) {
    // code to handle the second type of exception
} finally {
    // code that will always be executed, whether
    // or not an exception is thrown
}
```

In this example, the try block contains code that might throw an exception of ExceptionType1 or ExceptionType2. If one of these exceptions is thrown, the appropriate catch block will handle it. Regardless of whether an exception is thrown or not, the finally block will always be executed. This can be useful for releasing resources, closing files, or other cleanup tasks that need to be done regardless of the outcome of the code in the try block.

It's important to note that a finally block is optional and not required in every try-catch block. However, it can be very useful in situations where you need to perform cleanup tasks or ensure that certain code is always executed, regardless of whether an exception is thrown.

finally block

eazy bytes

In the finally block, the scanner object is closed to free up resources. The code in the finally blocks irrespective of exception occurs or not,

```java
import java.util.InputMismatchException;
import java.util.Scanner;

public class ExceptionDemo {

    public static void main(String[] args) {
        Scanner scanner = null;
        try{
            scanner = new Scanner(System.in);
            System.out.print("Enter a number: ");
            int input = scanner.nextInt();
            System.out.println(input);

        }catch (InputMismatchException e) {
            System.out.println("Please provide input in numeric values");
        }finally {
            if(scanner != null){
                scanner.close();
            }
        }

    }
}
```

try- with-resources statement

eazy bytes

One problem with exception handling while dealing with IO resources like Scanner, BufferedReader etc. is that developer has to write a finally block and make sure to close the resources. But what if Developer forgets or not closed properly ?

```
Scanner scanner = null;
try {
    scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int input = scanner.nextInt();
    System.out.println(input);
} catch (Exception ex) {
    // Exception handling logic
}finally {
    scanner.close();
}
```

To resolve this problem, in Java 7 try-with-resources is introduced. Here is the syntax & example of try-with-resources in Java. With the below code, when the try block exits, the Scanner object is automatically closed due to the use of try-with-resources. This ensures that the resource is properly closed, even if an exception is thrown. Behind the scenes, scanner.close() method will be automatically invoked by JVM.

```
try (resource_declaration) {
    // code that uses the resource
} catch (exception_declaration) {
    // exception handling code
}
```

```
try (Scanner scanner = new Scanner(System.in)){
    System.out.print("Enter a number: ");
    int input = scanner.nextInt();
    System.out.println(input);
} catch (Exception ex) {
    // Exception handling logic
}
```

From Java 7

try- with-resources statement

eazy bytes

- You can specify multiple resources as well inside the try-with-resources statement.
- A try-with-resources statement can itself have catch and finally clauses for other requirements inside the application.
- All resource reference variables should be final or effective final. So we can't perform reassignment with in the try block.
- Till Java 1.6, try block should be followed by either catch or finally block but from Java 7 we can have only try with resource block with out catch & finally blocks.

catch & finally are optional with try-with-resources

```
try (Scanner scanner = new Scanner(System.in) ){

        // code
}
```

Rules while handling exceptions

eazy bytes

try is mandatory

You cannot have a catch or finally without a try.

```
void display() {
    int a= 10;
    catch(Exception ex){

    }
}
```

Above method won't compile, because try block is missing

No code b/w try , catch & finally

```
void display() {
    try{
       int a= 10;
    }
     int j = 5;
    catch(Exception ex){
    }
}
```

Above method won't compile, because code between try & catch blocks is not allowed. The same applies for finally block as well

try should be followed by catch or finally or both

try block alone is not allowed & should be followed by either catch or finally or both. Below are the valid syntaxes

```
try {
}catch(){
}
```

```
try {
}finally(){
}
```

```
try {
}catch(){
}finally(){
}
```

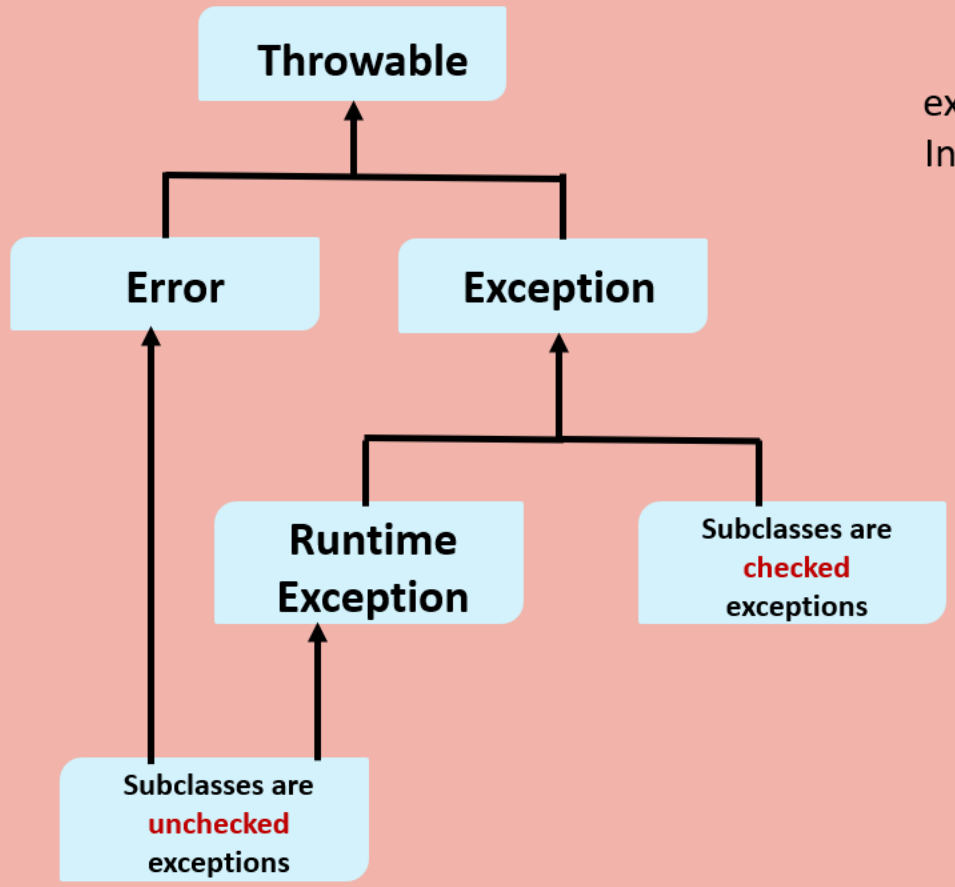
catch & finally are optional with try-with-resources

we can have only try with resource block with out catch & finally blocks. . Below are the valid syntaxes

```
try (Scanner scanner = new
Scanner(System.in) ){
   // code
}
```

The Exception Hierarchy

eazy bytes



The hierarchy of exceptions in Java is illustrated in figure, with all exceptions being subtypes of the Throwable class. Among these, the Error subclasses represent exceptional scenarios that the program is unable to handle, such as memory depletion. In such situations, little can be done except for displaying an error message to the user, indicating that a critical error has occurred.

Programmer-reported exceptions are subclasses of the class Exception. These exceptions fall into two categories:

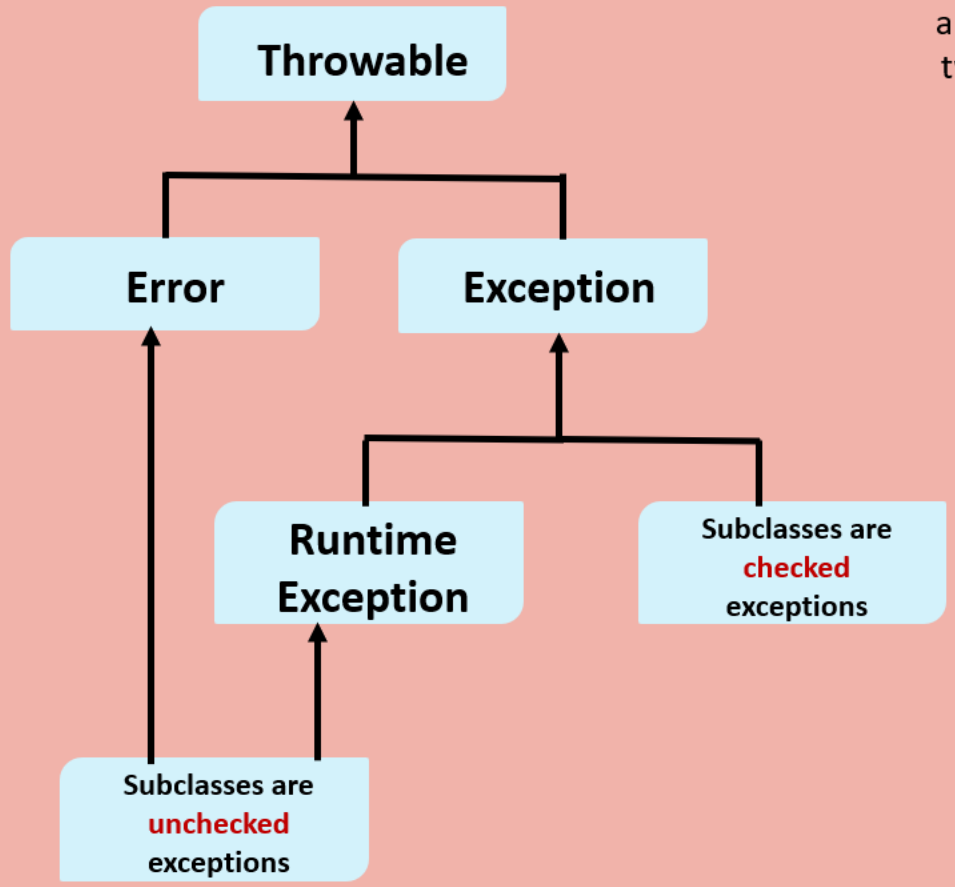
- Unchecked exceptions are subclasses of RuntimeException.
- All other exceptions are checked exceptions.

Subclasses of Error are also considered as Unchecked exceptions

Exceptions can be found in various packages beyond java.lang, depending on the package's supported functionality. For instance, the java.util package includes the runtime exception MissingResourceException, while the checked exception IOException is part of java.io.

The Exception Hierarchy

eazy bytes



By referring to the java.lang package API documentation, you will come to know that there are almost three dozen commonly used exception classes and a couple of dozen error classes. These two groups inherit all the methods from the java.lang.Throwable class and do not introduce any new methods. The frequently utilized methods in the java.lang.Throwable class are:

void printStackTrace(): It is used to print the stack trace of an exception or error that has occurred. The stack trace is a list of method calls that led to the point where the exception or error occurred. It is important to note that the printStackTrace() method should not be used in production code as it can potentially expose sensitive information and could be a security vulnerability. Instead, logging frameworks should be used.

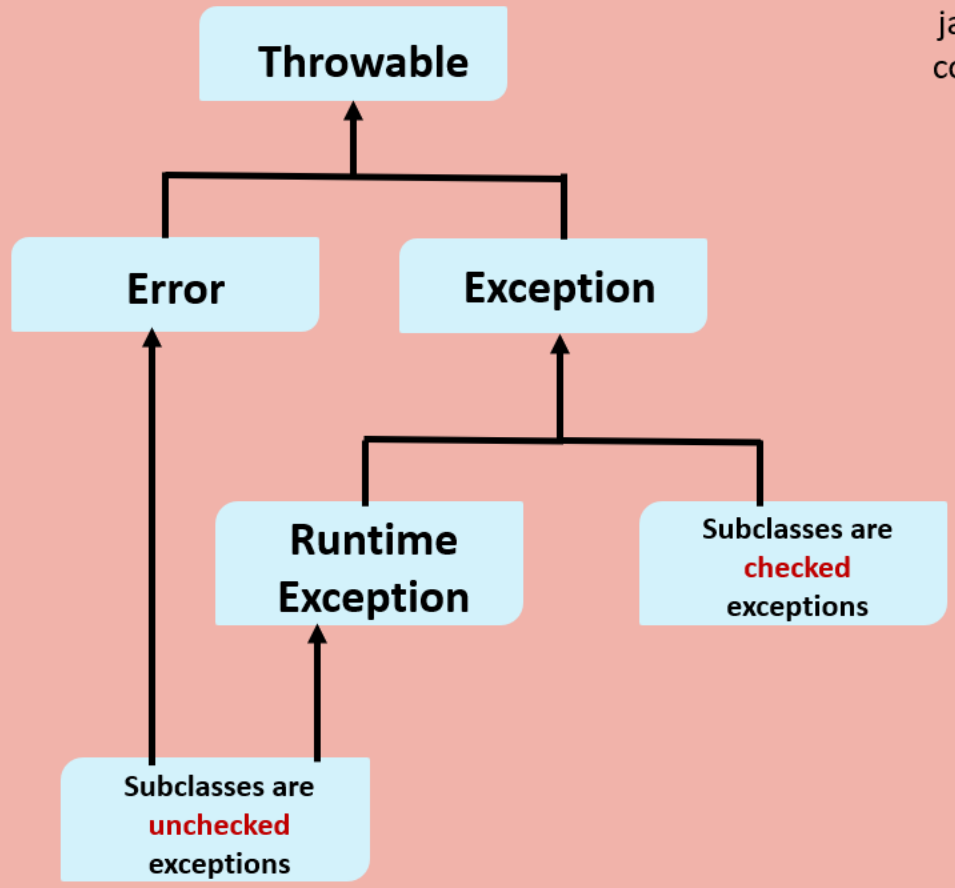
StackTraceElement[] getStackTrace(): It returns an array of StackTraceElement objects representing the method calls that led to the point where the exception or error occurred.

String getMessage(): It returns the message that often contains a user-friendly explanation of the reason for the exception or error.

Throwable getCause(): It returns the exception or error that caused the current exception to be thrown, or null if the cause is unknown or nonexistent. It is particularly useful when multiple exceptions are thrown in succession, as it allows you to trace the original cause of the problem.

The Exception Hierarchy

eazy bytes



The java.lang.Error class serves as the parent class for all errors in Java, and it inherits from the java.lang.Throwable class. Generally, an error is thrown by the Java Virtual Machine (JVM) and is considered to indicate severe issues that an application should not attempt to handle. As per the official documentation, here are some examples of such errors:

StackOverflowError: This is thrown when the memory allocated for the stack of the method calls is not enough to store another stack frame. This error is often caused by recursive functions that do not have proper base cases, causing them to call themselves indefinitely and eventually exhaust the call stack. It can also occur when a program relies heavily on nested function calls

NoClassDefFoundError: This is thrown when the JVM cannot find the definition of the class requested by the currently loaded class. This error can be caused by a variety of issues, such as a missing dependency, a class file that was not included in the distribution, or a class file that was corrupted during the build process.

OutOfMemoryError: This is thrown when the JVM runs out of memory and cannot clean the memory using garbage collection. This error typically occurs when a program attempts to allocate more memory than is available, or when there is a memory leak that causes the program to consume memory at an unsustainable rate.

The developers of the framework made an assumption that errors of this nature are typically irrecoverable and cannot be handled automatically by an application. This assumption has proven to be mostly accurate, which is why programmers generally do not catch these types of errors.

Checked and Unchecked Exceptions

eazy bytes

In Java, exceptions are categorized into two types: checked exceptions and unchecked exceptions.

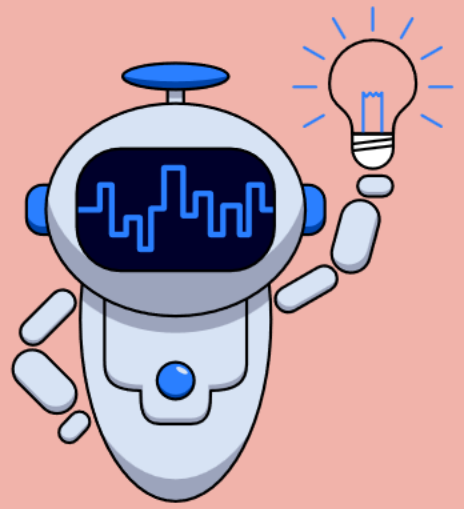
Checked exceptions are exceptions that the Java compiler requires the programmer to handle in the code. These exceptions are typically recoverable and are intended to be handled by the caller. Examples of checked exceptions include IOException, SQLException, and FileNotFoundException.

Java compiler will be angry at you and throw compilation error, if you write the risky code that may result in one of the checked exceptions but not handled it

VS

Unchecked exceptions are exceptions that the Java compiler does not require the programmer to handle in the code. These exceptions are typically caused by programming errors or other unexpected conditions that cannot be easily recovered from. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

Java compiler will not complain and continue with it's work, if you write the risky code that may result in one of the unchecked exceptions but not handled it. However, it is generally a good practice to handle these exceptions where possible to improve the robustness and reliability of the program.



Checked exceptions are those that do not have java.lang.RuntimeException in their hierarchy, and they must be handled or declared. The compiler verifies that these exceptions are either caught or mentioned in the method's throws clause.

Exceptions that inherit from java.lang.RuntimeException are referred to as unchecked or runtime exceptions. While all errors are also unchecked exceptions, they are typically not programmatically handled, so catching java.lang.Error descendants is unnecessary.

Checked and Unchecked Exceptions

eazy bytes

Here are some examples of Checked & Unchecked Exceptions:

IOException - thrown when an I/O operation fails, such as reading from or writing to a file.

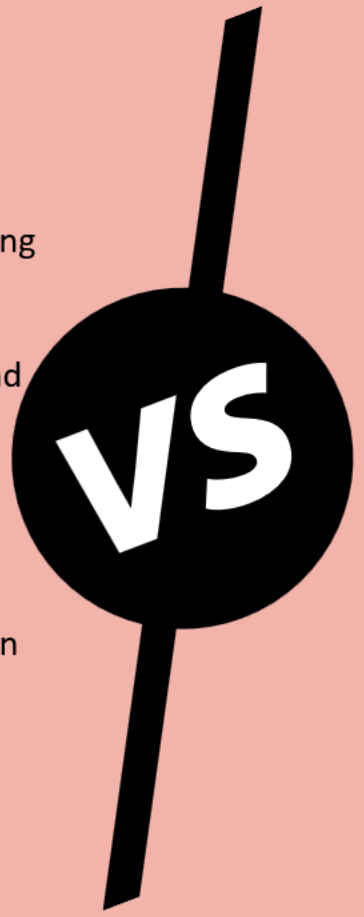
SQLException - thrown when an error occurs while accessing a database.

ClassNotFoundException - thrown when a class is not found at runtime.

ParseException - thrown when there is an error during the parsing of strings into date or number formats

MalformedURLException - thrown when there is an error in creating a URL object, such as when an invalid URL is provided

FileNotFoundException - thrown when a file is not found.



NullPointerException - thrown when a program attempts to use a null object reference.

ArrayIndexOutOfBoundsException - thrown when an array is accessed with an illegal index.

ArithmeticException - thrown when an arithmetic operation produces an error, such as division by zero.

IllegalArgumentException - thrown when an illegal argument is passed to a method.

IllegalStateException - thrown when the state of an object is incompatible with the requested operation.

ClassCastException - thrown when a program attempts to cast an object to an incompatible class.

throws keyword

eazy bytes

Scenario 1 :

Suppose you, as a developer, have written a method that has the potential to throw an exception. However, you do not wish to handle the exception yourself and want the caller of the method to handle it instead. How can you convey this information?

Scenario 2:

Suppose you need to use a method from a class that you did not create. This method performs a potentially risky operation that may not work at runtime. You need to be aware of the fact that the method you are calling is risky and take appropriate measures to handle any potential failures. How can you determine if a method may throw an exception?

throws keyword

eazy bytes

In Java, the throws keyword is used in a method signature to declare the exceptions that can be thrown by the method. When a method can potentially throw a exception, it must either handle the exception using a try-catch block or declare the exception using the throws keyword. The throws keyword is followed by a comma-separated list of exception classes that the method may throw. For example, consider the following method signature:

```
public double divide(String num1, String num2)
        throws NumberFormatException, ArithmeticException {

    // method logic

}
```

This method takes two String parameters, num1 and num2, and returns their quotient as a double. If either num1 or num2 cannot be parsed into a valid double, a NumberFormatException will be thrown. If num2 is 0, an ArithmeticException with the message "Division by zero" will be thrown.

The method declares both of these exceptions in its throws clause, indicating that it may throw them during execution. Any code that calls this method must either handle these exceptions using a try-catch block or declare them in its own throws clause.

throw keyword

eazy bytes

In Java, the throw keyword is used to explicitly throw an exception. It is followed by an instance of the Throwable class or one of its subclasses, which represents the exception being thrown. We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section. Here's an example of using the throw keyword to throw an exception:

```
Scanner scanner = new Scanner(System.in);
System.out.println("Please enter the numerator value");
String num1 = scanner.next();
System.out.println("Please enter the denominator value");
String num2 = scanner.next();
Division division = new Division();
if(num2.equals("0")) {
    throw new ArithmeticException("Division by zero");
}
double output = division.divide(num1, num2);
System.out.println(output);
```

In this example, the divide method throws an ArithmeticException if the denominator is zero. The throw keyword is used to create a new instance of the ArithmeticException class and throw it.

throw & throws

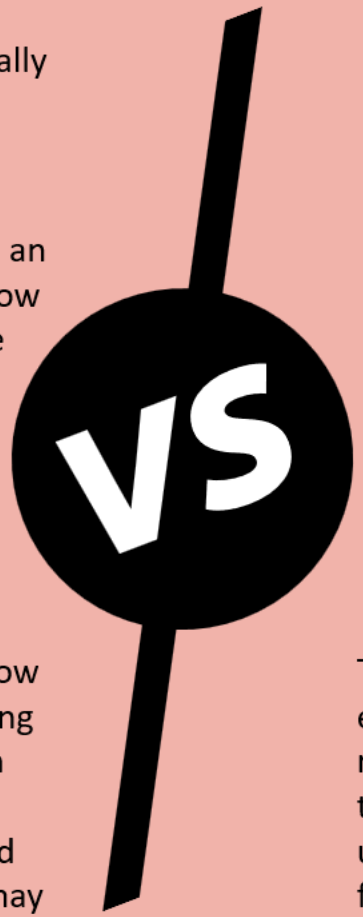
eazy bytes

The keywords throw and throws in Java have several differences. The following is a list of the distinctions between them:

The throw keyword is employed within a function to manually throw an exception when an error or unexpected situation arises.

When using the throw keyword in Java, the syntax requires an instance of the exception that needs to be thrown. The throw keyword is followed by the instance variable containing the exception object.

Using throw keyword, we can only throw one exception at a time.



The throws keyword is utilized in the method signature to indicate that the method may throw exceptions due to certain statements or actions within its body.

When using the throws keyword in Java, the syntax necessitates the class names of the exceptions that may be thrown by the method. The throws keyword is followed by the class names of the exceptions separated by commas.

Using throws keyword we can declare multiple exceptions, separated by commas. If any of the declared exceptions match the exception that occurs during function execution, it is automatically thrown.

Only unchecked exceptions can be propagated using the throw keyword. This implies that throwing a checked exception using throw alone is not allowed, and it will result in a compilation error. To throw a checked exception using throw, a throws declaration must also be included to inform the compiler and calling code about the checked exception that the method may throw.

The throws keyword can be used to declare both checked and unchecked exceptions, but it can only be used to propagate checked exceptions. This means that when a method declares a checked exception using throws, the calling code must handle or propagate the exception to the next level using a try-catch block or another throws declaration. If the calling code fails to handle or propagate the checked exception, a compilation error will occur.

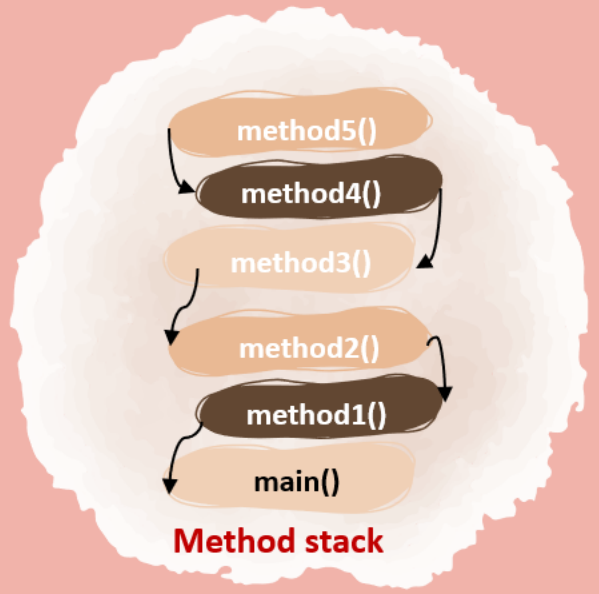
Exception Propagation

eazy bytes

When an exception is thrown from a method and is not caught within that method, it is propagated to the calling method. This process continues until either the exception is caught by a catch block or it reaches the top-level method (main method), where it is handled by the default exception handler.

During the exception propagation, the call stack is searched for the nearest catch block that can handle the exception. If no catch block is found, the exception is propagated up to the calling method. If the exception is not caught by any method, it eventually reaches the top-level method, where it is either handled or causes the program to terminate.

It's worth noting that checked exceptions must be handled or declared in the method signature, otherwise, the compiler will throw an error. Unchecked exceptions, on the other hand, are not required to be handled or declared.



If an exception occurs in method5() & if none of the methods in the call stack didn't handle the exception, then the exception will keep propogate till it reaches the end of the method stack which has main method()

Nested try block

eazy bytes

In Java, a nested try block is a try-catch block that is placed inside another try-catch block. The purpose of a nested try block is to handle exceptions that may occur within a specific block of code, while still being able to handle exceptions that occur in the outer block of code. The syntax for a nested try block is as follows:

```
// main try block
try {
    statement 1;
    statement 2;
    // try-catch block within another try block
    try {
        statement 3;
        statement 4;
        // try-catch block within nested try block
        try {
            statement 5;
            statement 6;
        } catch (Exception e2) {
            // exception message
        }
    } catch (Exception e1) {
        // exception message
    }
} catch (Exception e3) {
    // catch block of parent (outer) try block
    // exception message
}
```

In this code, there are three levels of exception handling. The outermost try block handles any exceptions that occur in the statements within it. The second try block is nested inside the outermost try block and handles any exceptions that occur in statements 3 and 4. The third try block is nested inside the second try block and handles any exceptions that occur in statements 5 and 6.

If an exception is thrown in statement 5 or 6, it is caught by the innermost catch block (catch (Exception e2)). If an exception is thrown in statement 3 or 4, it is caught by the middle catch block (catch (Exception e1)). If an exception is thrown in statement 1 or 2, it is caught by the outermost catch block (catch (Exception e3)).

Custom Checked Exception

eazy bytes

Java provides a comprehensive set of general exceptions that cover most common programming scenarios. Nevertheless, there are situations where it's necessary to add custom exceptions to complement these standard ones.

The main reasons for creating custom exceptions are twofold. Firstly, to handle exceptions that are unique to the specific business logic and workflow of the application. Custom exceptions help developers and end-users to pinpoint the exact cause of the problem, which aids in resolving issues quickly. Secondly, custom exceptions can be used to catch and handle specific subsets of existing Java exceptions in a more targeted and efficient manner.

Here's an example of creating a custom checked exception InvalidAgeException that can be used for invalid ages in a Java program.

```
public class InvalidAgeException extends Exception {

    public InvalidAgeException() {
        super();
    }

    public InvalidAgeException(String message) {
        super(message);
    }

}
```

Custom Unchecked Exception

eazy bytes

To create a custom unchecked exception, we need to extend RuntimeException. Here's an example of creating a runtime custom exception to throw when the denominator is zero and using the same inside the program.

```
public class DivideByZeroException extends RuntimeException {

    public DivideByZeroException() {
        super();
    }

    public DivideByZeroException(String message) {
        super(message);
    }

}
```

final, finally and finalize

eazy bytes

In Java, "final," "finally," and "finalize" are three distinct jargons that have different meanings and usage:

final

"final" is a keyword used to create a constant value that cannot be changed. When a variable or method is marked as "final," its value or behavior cannot be altered, making it a constant. For example, "final int x = 5;" creates a constant integer value of 5 that cannot be changed later in the program. Similarly, when final is mention inside the class definition, then the class can't be inherited

```
public final class Example
{
    final int x =5;
}
```

finally

"finally" is a keyword used in a try-catch-finally block in Java. It is used to specify a block of code that will be executed regardless of whether an exception is thrown or caught in the try or catch blocks. This ensures that certain code is always executed, such as closing a database connection or releasing a lock, even if an exception occurs.

```
try {
}catch(Exception ex){
}finally{
}
```

finalize

"finalize" is a method in Java that is called by the garbage collector when an object is no longer being used and is about to be removed from memory. It is a method that can be overridden by a class to perform cleanup operations or release resources before the object is destroyed. However, it is important to note that finalize() is not guaranteed to be called and should not be relied upon for critical cleanup operations.

```
@Override
protected void finalize()
    throws Throwable {


}
```