**Exception handling** in Java is a mechanism that allows a program to deal with runtime errors, so it can continue running or fail gracefully, without abruptly crashing. Java uses a try-catch block to handle exceptions, which are events that disrupt the normal flow of the program's execution.

The **Exception** class in Java is a **subclass of Throwable**, which is the superclass of all errors and exceptions in Java. These objects contain the stack trace and all information regarding the exception.

In Java, java.lang.Exception class is the superclass of all exception objects. These objects contain the stack trace and all information regarding the exception.

The runtime environment (JRE) produces and throws the exception object whenever an exception occurs. The instant after an exception object is thrown, the program stops the execution. If the exception is not taken care of, then it is propagated to the calling environment. It can be either a runtime system or a calling method.

class Except {

public static void divide(int x, int y) {

int z = x / y;

System.out.println(z);

}

public static void main(String[] args) {

divide(10, 0);

}

}

In the above case, an exception is generated inside the divide() method. Since the divide() method doesn't take care of it, the exception propagates to its calling environment, which is the main() method. The main() method also doesn't take care of it, and the exception propagates to its calling environment, which is the runtime.

When the runtime environment (JRE) receives the exception, it terminates the program and shows the stack trace.

When something goes wrong during program execution (like a division by zero or trying to access a file that doesn't exist), an exception is thrown. Exception handling allows you to catch these errors and handle them appropriately.

Type of Error Description Examples

Compile-time Errors Errors in code structure or syntax that prevent the code from compiling. Missing semicolon, mismatched braces, wrong variable type.

Checked Exceptions Exceptions that can occur during program execution, which the compiler forces you to handle. IOException, SQLException, FileNotFoundException

Runtime Errors Errors that happen during program execution (often due to invalid logic or inputs). NullPointerException, ArrayIndexOutOfBoundsException

In short:

**Compile-time errors** are related to syntax mistakes like missing semicolons, incorrect keywords, or other basic mistakes that prevent the program from compiling.

**Checked exceptions**: These are exceptions that are checked at compile-time. The compiler forces you to handle them either by using a try-catch block or declaring the exception using throws clause.

Checked exceptions require explicit handling. If you don't handle them, the program will not compile.

Example: IOException, SQLException, ClassNotFoundException

If you have a **checked exception**, the compiler **forces you** to either:

1. **Handle it** with a try-catch block
2. **Declare it** in the method signature with throws

If you don’t do either of those, **the code won’t compile.**



**A screenshot of a computer

AI-generated content may be incorrect.**

**Unchecked Exceptions:** These are exceptions that are not checked at compile-time. They usually indicate programming errors (e.g., accessing an array index that doesn’t exist). Unchecked exceptions don't require explicit handling, but it’s good practice to handle them if they could arise

Example: NullPointerException, ArrayIndexOutOfBoundsException.

A screenshot of a computer program

AI-generated content may be incorrect.

**Try** Block: The code that might throw an exception is placed inside a try block. If an exception occurs, it is "caught" by the catch block.

**Catch** Block: This is where you handle the exception. The catch block is only executed if an exception is thrown inside the corresponding try block.In a catch block, specific exceptions are preferred rather than general exceptions or unspecific exceptions.

can have multiple catch blocks for single try block but the catch blocks with specified exceptions like **catch (ArithmeticException e2) block** ..etc should come **before** the more general **catch (Exception e1)** block.

If an exception arises from the try block, the matching catch block gets executed. In the try block, the lines of code after the line which caused exception, are ignored.

If a catch block handles the exception, then the execution continues from the code placed after the try-catch block.

public static void divide(int x, int y) {

try {

int z = x/y; // If an exception occurs here, the control jumps to the first matching catch block

System.out.println(z); // Execution of this line will be skipped

}

catch(ArrayIndexOutOfBoundsException e) {

System.out.println("Index not found");

}

catch(ArithmeticException e) { // This is the matching exception handler

System.out.println("The divisor should not be zero");

}

System.out.println("Method execution ends"); // Program execution continues from here

}

If the suitable catch block is not found, the exception remains unhandled and propagated.

If no exception gets thrown from the try block, their subsequent catch blocks are ignored.

A catch block that can handle objects of Exception class can catch all the exceptions.

This below block should always be the last catch block in the catch sequence.

**catch(Exception e) {**

**// Code for handling exception**

**}**

If a similar exception handling logic is required for multiple exceptions, a multi-catch block can be used. From Java 7 onwards one or more exceptions can be handled by a single catch block.

catch(Exceptiontype1 | Exceptiontype2 | Exceptiontype3 e) {

// Code to handle exception goes here

}

**Finally Block**: The finally block is always executed, whether an exception is thrown or not. It is used for cleaning up resources like closing files or database connections.

public class ExceptionHandlingDemo {

public static void main(String[] args) {

int[] intArray = { 1, 2, 3, 4, 5 };

int intVar = 0;

try {

intVar = intArray[5];

System.out.println("This is the value of intVar:=" + intVar);// 1

} catch (ArrayIndexOutOfBoundsException ec) {

System.out.println("An exception has occurred here:="

+ ec.getMessage());// 2

} finally {

System.exit(-1);

System.out.println("This is from with in the finally block");// 3

}

System.out.println("This will be a part of response");// 4

}

}

when System.exit(-1) is used in finally block, JVM automatically terminates the program and line //4 will not be executed.

**Throw**: The throw keyword is used to explicitly throw an exception from your code.

**Throws**: The throws keyword is used to declare that a method may throw an exception.

----------------------------------------------------------

Basic Syntax of Exception Handling

try {

// Code that may cause an exception

} catch (ExceptionType e) {

// Code to handle the exception

} finally {

// Code that will always execute, regardless of an exception

}

Example of Exception Handling:

public class Main {

public static void main(String[] args) {

try {

int result = 10 / 0; // This will cause an ArithmeticException

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

// Handling ArithmeticException (e.g., divide by zero)

System.out.println("Error: Cannot divide by zero.");

} finally {

// This block will always run

System.out.println("This is the 'finally' block.");

}

}

}

Explanation of the Code:

**Try Block**: The division operation 10 / 0 is inside the try block. This will throw an ArithmeticException because dividing by zero is not allowed.

**Catch Block**: The catch block catches the ArithmeticException and prints a user-friendly error message: "Error: Cannot divide by zero."

Finally Block: Regardless of whether an exception occurs or not, the finally block will always run. In this case, it prints "This is the 'finally' block."

**The throw and throws Keywords**

Throw: You use throw to explicitly throw an exception. It is followed by an instance of an exception and helps in writing custom messages when exception is thrown.

public class Main {

public static void main(String[] args) {

try {

throw new Exception("Something went wrong.");

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

Java allows to explicitly generate or throw exceptions using the throw keyword:

Exception e = new Exception();

throw e;

Any Throwable object type can be thrown.

Exception messages can be custumized:

throw new Exception("The divisor should not be zero");

Having exceptions with custom messages increases the readability of our applications.

Below is an example…

public static void divide(int x, int y) {

try {

if(y == 0)

throw new Exception("The divisor should not be zero"); //throwing exception here

int z = x/y;

System.out.println(z);

}

catch(Exception e) {

System.out.println(e.getMessage());

}

}

The above code will generate an exception with the given message if the condition is satisfied.

**Throws**: The throws keyword is used in a method declaration to indicate that the method may throw certain exceptions, and it is up to the calling code to handle them.

example 1:

public class Main {

public static void main(String[] args) {

try {

methodThatThrowsException();

} catch (Exception e) {

System.out.println("Caught exception: " + e.getMessage());

}

}

// Declaring that this method can throw an exception

public static void methodThatThrowsException() throws Exception {

throw new Exception("An error occurred!");

}

}

example 2:

public static void divide(int x, int y) throws Exception {

if(y == 0)

throw new Exception("The divisor should not be zero"); //throwing exception to main method from where this method is called.

int z = x/y;

System.out.println(z);

}

public static void main(String args[]) {

try {

divide(10, 0);

}

catch(Exception e) {

System.out.println(e.getMessage());

}

}

**Chained Exceptions**

In Java, you can chain exceptions by passing one exception as the cause of another exception. This is done by passing the original exception to the constructor of a new exception.

try {

// Code that throws an exception

throw new ArithmeticException("This is the cause");

} catch (ArithmeticException e) {

throw new RuntimeException("Exception occurred while processing", e); // Chaining exception

}

Best Practices for Exception Handling

Handle Specific Exceptions: Catch the most specific exceptions first. For example, catch FileNotFoundException before IOException.

Don’t Overuse Exception Handling: Exceptions should be used for exceptional conditions, not for regular control flow. Avoid using exceptions for common situations (e.g., using try-catch just to check if a file exists).

Log Exceptions: In production environments, it’s a good practice to log the exception details (including the stack trace) so that you can debug issues later.

Don’t Swallow Exceptions: Avoid catching exceptions without handling them properly. For example, printing the stack trace or rethrowing the exception with additional context can be more useful than just ignoring it.

Conclusion

Exception handling in Java is essential for making your programs robust and user-friendly. By using try-catch blocks, you can handle errors gracefully and ensure your program doesn’t crash unexpectedly. Java also provides finally blocks for cleaning up resources and throws/throw for declaring and explicitly throwing exceptions.

what are the methods in exception class

The Exception class in Java is a subclass of Throwable, which is the superclass of all errors and exceptions in Java. The Exception class itself provides several useful methods that help in handling exceptions. Here are the most commonly used methods in the Exception class:

1. getMessage()

Description: Returns a detailed message about the exception.

Return Type: String

Example:

java

Copy

try {

throw new Exception("Something went wrong");

} catch (Exception e) {

System.out.println(e.getMessage()); // Outputs: Something went wrong

}

2. printStackTrace()

Description: Prints the stack trace of the exception to the standard error stream (System.err). The stack trace is a list of method calls that were on the call stack when the exception was thrown.

Return Type: void

Example:

try {

throw new Exception("An error occurred");

} catch (Exception e) {

e.printStackTrace(); // Prints the stack trace to the console

}

3. toString()

Description: Returns a string representation of the exception, which includes the class name and the result of getMessage() if provided.

Return Type: String

Example:

try {

throw new Exception("Custom exception message");

} catch (Exception e) {

System.out.println(e.toString()); // Outputs: java.lang.Exception: Custom exception message

}

4. getCause()

Description: Returns the cause of the exception (i.e., another Throwable that caused this exception). This is useful in cases where exceptions are wrapped inside other exceptions (like in try-catch blocks).

Return Type: Throwable

Example:

try {

throw new Exception("Outer exception", new NullPointerException("Inner exception"));

} catch (Exception e) {

System.out.println(e.getCause()); // Outputs: java.lang.NullPointerException: Inner exception

}

5. getStackTrace()

Description: Returns an array of StackTraceElement objects that represents the stack trace for the exception. This can be used for more detailed programmatic analysis of the stack trace.

Return Type: StackTraceElement[]

Example:

try {

throw new Exception("Stack trace example");

} catch (Exception e) {

StackTraceElement[] stackTrace = e.getStackTrace();

for (StackTraceElement element : stackTrace) {

System.out.println(element);

}

}

6. initCause(Throwable cause)

Description: Initializes the cause of the exception, which can be another Throwable that caused this exception.

Return Type: Throwable

Example:

Exception outerException = new Exception("Outer exception");

Exception innerException = new Exception("Inner exception");

outerException.initCause(innerException);

System.out.println(outerException.getCause()); // Outputs: java.lang.Exception: Inner exception

7. fillInStackTrace()

Description: This method is used to fill in the execution stack trace information. By default, the stack trace is filled in when an exception is thrown. This method can be used to fill in the stack trace manually.

Return Type: Throwable

Example:

try {

throw new Exception("Filling stack trace manually");

} catch (Exception e) {

e.fillInStackTrace(); // Manually fills the stack trace

e.printStackTrace(); // Prints the updated stack trace

}

8. getLocalizedMessage()

Description: Returns a localized description of the exception. If a message is set during exception creation, this will return that message, but it may be localized based on the system's locale.

Return Type: String

Example:

try {

throw new Exception("Localized message");

} catch (Exception e) {

System.out.println(e.getLocalizedMessage()); // Outputs: Localized message

}

9. addSuppressed(Throwable exception)

Description: Adds a suppressed exception to the list of exceptions that were suppressed in order to allow multiple exceptions to be reported together (for example, when closing resources).

Return Type: void

Example:

try {

Exception e1 = new Exception("Main exception");

Exception e2 = new Exception("Suppressed exception");

e1.addSuppressed(e2);

throw e1;

} catch (Exception e) {

e.printStackTrace(); // Outputs the main exception and suppressed exception

for (Throwable t : e.getSuppressed()) {

System.out.println("Suppressed: " + t.getMessage());

}

}

Summary:

getMessage(): Gets the message string describing the exception.

printStackTrace(): Prints the stack trace to the standard error stream.

toString(): Returns a string representation of the exception.

getCause(): Gets the cause of the exception.

getStackTrace(): Gets the stack trace as an array of StackTraceElement.

initCause(): Initializes the cause of the exception.

fillInStackTrace(): Fills the stack trace with the current stack information.

getLocalizedMessage(): Returns the localized message for the exception.

addSuppressed(): Adds a suppressed exception to the list of exceptions.

These methods are helpful in exception handling and provide ways to retrieve detailed information about an exception, track the flow of execution, and manage multiple exceptions.

Checked Exceptions in Java

In Java, checked exceptions are exceptions that are checked at compile-time. This means that if a method can throw a checked exception, the compiler forces you to handle it—either by using a try-catch block or by declaring the exception in the method signature using the throws keyword. If you don't handle these exceptions correctly, your program won't compile.

Why Checked Exceptions?

Checked exceptions are typically used to indicate recoverable conditions in the program. These are scenarios where the program might be able to recover from the error, or the programmer can provide a way to handle the error gracefully.

For example:

If you are working with files, the file may not exist, or there may be a problem reading it. In such cases, an exception (IOException) could be thrown, and the program can handle it (for example, by asking the user to select a different file).

Checked Exceptions vs Unchecked Exceptions

Checked exceptions: Must be handled or declared by the programmer (e.g., IOException, SQLException).

Unchecked exceptions: These are usually programming errors that do not need to be explicitly handled (e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException).

How Checked Exceptions Work

1. Handling Checked Exceptions

When a method is capable of throwing a checked exception, you must handle it by using a try-catch block or declare it in the method's signature using throws.

Example 1: Handling a Checked Exception Using try-catch

Let's say we are reading from a file. The FileReader class in Java throws a FileNotFoundException, which is a checked exception. We need to handle it using try-catch.

import java.io.FileReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try {

// Attempt to read from a file

FileReader file = new FileReader("nonexistentFile.txt");

} catch (IOException e) {

// Handle the exception

System.out.println("Error: The file could not be found.");

}

}

}

In this example:

The FileReader constructor throws a FileNotFoundException (which is a subclass of IOException).

The catch block catches that exception and prints a message.

If we don't catch or declare the exception, the program will not compile.

Example 2: Declaring a Checked Exception Using throws

Alternatively, if you don't want to handle the checked exception inside the method, you can declare it using the throws keyword in the method signature. This allows the calling code to handle the exception.

import java.io.FileReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try {

// Calling a method that may throw an exception

readFile();

} catch (IOException e) {

// Handle the exception

System.out.println("Error: The file could not be found.");

}

}

// Declare that this method can throw an IOException

public static void readFile() throws IOException {

FileReader file = new FileReader("nonexistentFile.txt");

}

}

In this example:

The readFile method declares that it throws an IOException by using throws.

The main method calls readFile and catches the IOException.

This way, readFile doesn't need to handle the exception, but it declares that it may throw an exception.

2. List of Common Checked Exceptions

Here are a few common examples of checked exceptions in Java:

IOException: Thrown when an I/O operation fails or is interrupted (e.g., when reading from a file, or if there is a problem with network connections).

Example:

throw new IOException("I/O operation failed");

SQLException: Thrown when there is an issue with database access.

Example:

throw new SQLException("Database error");

ClassNotFoundException: Thrown when a class is not found during runtime (e.g., when loading a class using reflection).

Example:

throw new ClassNotFoundException("Class not found");

FileNotFoundException: A subclass of IOException, thrown when trying to open a file that doesn't exist.

Example:

throw new FileNotFoundException("File not found");

Why Do We Need to Handle Checked Exceptions?

Java's design choice to require handling of checked exceptions allows developers to address potentially recoverable errors. For example:

When dealing with file I/O, you might want to catch and handle the scenario where a file doesn't exist, or the user doesn't have permission to access the file.

If you're working with a database, handling SQLException ensures that issues like connection failures, invalid queries, or constraint violations don't crash the program. Instead, you can handle the error and try to recover or inform the user.

If Java did not require handling checked exceptions, many errors in programs (especially related to external systems like file systems or databases) could silently fail or cause unhandled crashes.

Example of Multiple Checked Exceptions

A method may throw multiple checked exceptions. In such cases, you can either handle them separately in multiple catch blocks or combine them in a single catch block if they are of the same type or subclass.

import java.io.FileReader;

import java.io.IOException;

import java.sql.SQLException;

public class Main {

public static void main(String[] args) {

try {

// Simulating a file operation and a database operation

FileReader file = new FileReader("file.txt");

// Some database operation that might throw SQLException

throw new SQLException("Database connection issue");

} catch (IOException | SQLException e) {

// Handle both IOException and SQLException

System.out.println("Error occurred: " + e.getMessage());

}

}

}

In the example above:

The method may throw both an IOException and a SQLException.

We catch both exceptions in a single catch block (Java 7 and later allows this).

Summary

Checked exceptions are exceptions that must be handled explicitly in Java.

The compiler forces you to either use a try-catch block or declare the exception in the method signature with throws.

Checked exceptions typically represent recoverable errors, such as file errors, database errors, or network errors.

If you don't handle a checked exception, your program will not compile, making it a compile-time error.

Common checked exceptions include IOException, SQLException, and ClassNotFoundException.

Would you like to explore more examples or dive deeper into handling specific checked exceptions?

You said:

what i thought about compile time exception is that if we miss giving semicolon at the end of some initilaization of variable or anything

Ah, I see where you're coming from! The type of errors you're referring to—like missing a semicolon, mismatched brackets, or incorrect syntax—are actually compile-time errors, but they are not the same as checked exceptions. Let me clarify the difference:

Compile-time Errors (Syntax Errors)

These are errors that occur when the compiler cannot compile the code because it violates language syntax. These errors are typically related to mistakes in the structure of your code, such as:

Missing semicolons (;)

Mismatched parentheses (), curly braces {}, or square brackets []

Incorrect variable declarations

Misspelled keywords or identifiers

Incorrect method signatures or missing return types

Example of Compile-time Error:

public class Main {

public static void main(String[] args) {

int a = 5 // Missing semicolon, compile-time error!

System.out.println(a);

}

}

In the above code:

The missing semicolon (;) after int a = 5 causes a compile-time error.

This type of error is caught by the compiler when you try to compile the program.

The compiler cannot generate bytecode until these errors are fixed.

Checked Exceptions (Runtime vs Compile-time)

On the other hand, checked exceptions are related to runtime situations that the compiler can anticipate, such as trying to read from a file that may not exist or connecting to a database that may not be available. These are not syntax errors but rather conditions that might happen during the execution of the program.

Checked exceptions: Errors the compiler forces you to handle because it recognizes that there are situations where they might happen, like file I/O, network errors, or database access errors.

Example of a Checked Exception:

import java.io.FileReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try {

FileReader file = new FileReader("nonexistentFile.txt"); // Throws a checked exception

} catch (IOException e) {

System.out.println("An error occurred: " + e.getMessage()); // Handling checked exception

}

}

}

Here:

FileNotFoundException (a subclass of IOException) is a checked exception. The compiler forces you to handle it because opening a file might fail if the file doesn't exist or isn't accessible.

If you don't handle it using a try-catch block or declare it with throws, the code won't compile.

Summary of Differences

Type of Error Description Examples

Compile-time Errors Errors in code structure or syntax that prevent the code from compiling. Missing semicolon, mismatched braces, wrong variable type.

Checked Exceptions Exceptions that can occur during program execution, which the compiler forces you to handle. IOException, SQLException, FileNotFoundException

Runtime Errors Errors that happen during program execution (often due to invalid logic or inputs). NullPointerException, ArrayIndexOutOfBoundsException

In short:

Compile-time errors are related to syntax mistakes like missing semicolons, incorrect keywords, or other basic mistakes that prevent the program from compiling.

Checked exceptions are situations where errors can occur at runtime, but they are not syntax errors—they are more about external conditions (like file availability, database connection, etc.). The compiler makes sure you're aware of them and requires that you handle them.

What Happens if You Skip the try-catch or throws?

If you try to skip exception handling when working with code that can throw a checked exception, the compiler will throw an error and prevent your program from compiling.

For example:

import java.io.FileReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

// This will cause a compile-time error because IOException is not handled or declared

FileReader file = new FileReader("existentFile.txt");

}

}

The compiler will produce an error message like this:

Unreported exception IOException; must be caught or declared to be thrown

This means the compiler insists/forces that you must handle or declare the IOException.

Can You "Ignore" It in Practice?

If you are certain that there will be no issues and you're willing to take the risk of not handling the exception, there is a workaround—but this is not recommended in production code.

You can use a try-catch block with an empty catch (which will catch the exception but do nothing), though this is usually considered bad practice because it hides potential problems.

import java.io.FileReader;

import java.io.IOException;

public class Main {

public static void main(String[] args) {

try {

// Try to open the file without doing anything in case of exception

FileReader file = new FileReader("existentFile.txt");

} catch (IOException e) {

// Doing nothing (this is not recommended)

}

}

}

However, this still handles the exception—you're just not reacting to it, which isn't good practice. It's better to either handle the exception properly or declare it with throws if you want to pass the responsibility to another method.

Summary

Even if you're 100% sure there won't be any issue accessing the file, Java's compiler still requires that you handle or declare the checked exception.

This is because Java enforces robustness and doesn't allow you to skip handling checked exceptions, as doing so could lead to unhandled problems in other environments or future code changes.

The best practice is to either handle the exception properly (with try-catch) or declare it in the method signature (throws).

is number format exception is runtime exception

Yes, NumberFormatException is a runtime exception in Java.

Key Points:

NumberFormatException is a subclass of IllegalArgumentException, which itself is a subclass of RuntimeException.

Since it is a subclass of RuntimeException, NumberFormatException is considered an unchecked exception.

Unchecked Exceptions:

Unchecked exceptions are not required to be declared in a method's throws clause or handled with a try-catch block.

They can occur during the normal execution of the program and typically represent programming bugs, such as invalid arguments or illegal operations.

Example of NumberFormatException:

public class Main {

public static void main(String[] args) {

try {

// This will throw a NumberFormatException because "abc" is not a valid integer

int number = Integer.parseInt("abc");

} catch (NumberFormatException e) {

System.out.println("Caught a NumberFormatException: " + e.getMessage());

}

}

}

Output:

lua

Caught a NumberFormatException: For input string: "abc"

Why NumberFormatException is an unchecked exception:

It typically arises when the program tries to convert an invalid string into a number, like parsing "abc" into an integer.

The fact that it's unchecked means that you aren't forced by the compiler to handle it with try-catch or throws, although you can still handle it if you want to.

Summary:

NumberFormatException is a runtime exception (unchecked exception).

It doesn't need to be explicitly handled (though you can if necessary), as it extends RuntimeException.

Below is the table with methods from Throwable class which are used to get exception details:

Method Description

String getMessage() Detail of exception description is returned.

void printStackTrace() Detail of stack trace is returned.

String toString() A short description is returned.

Check how you can use the above methods:

System.out.println("Exception message: " + throwable.getMessage());

throwable.printStackTrace();

System.out.println("Description of the exception: " + throwable.toString());

Note: The methods mentioned above have been demonstrated in the subsequent tryout.

* In Java, a method can **declare** that it might throw certain exceptions using the throws keyword in its signature. This is typically used with **checked exceptions** (those that extend Exception), such as IOException, SQLException, etc.
* **Unchecked exceptions** (like ArithmeticException, NullPointerException, etc.), which are subclasses of RuntimeException, do **not require a throws declaration** in the method that throws them. Even though they are not declared in the method signature, they can still be thrown and propagated upwards in the call stack. This is a key difference from checked exceptions.

**Behavior of the display() Method:**

* The display() method in your code **throws an ArithmeticException** (which is an unchecked exception).
* It **does not need** to declare throws ArithmeticException or any other exception because unchecked exceptions do not require a declaration.
* When throw new ArithmeticException(); is executed in display(), the exception propagates back to the method that called display(), which is main() in this case.
* The exception is then caught by the runtime, and since main() doesn't handle it, the program terminates.

**Example:**

Let’s compare with a method that uses a **checked exception** and requires a throws declaration.

**Checked Exception (requires throws):**

public class Test {

public static void main(String[] args) throws IOException { // 1

readFile(); // 2

}

public static void readFile() throws IOException { // 3

throw new IOException(); // 4

}

}

In this example:

* readFile() throws a checked exception (IOException), so it must declare throws IOException in its method signature.
* The main() method must also declare throws IOException if it wants to propagate the exception or catch it.

**Unchecked Exception (like ArithmeticException):**

For your code:

public class TestingCode {

public static void display() {

System.out.println("Inside Display");

throw new ArithmeticException(); // No `throws` declaration needed

}

public static void main(String[] args) {

display(); // Calls display()

System.out.println("Inside Main"); // This line is never reached

}

}

Here:

* display() throws an unchecked exception (ArithmeticException).
* It does **not need** to declare throws ArithmeticException in the method signature.
* The exception propagates to main(), but main() doesn't handle it with a try-catch, so the program terminates with an uncaught exception.

**Key Takeaway:**

* **Checked exceptions** (like IOException, SQLException) require a throws declaration in the method signature if they are thrown and not caught.
* **Unchecked exceptions** (like ArithmeticException, NullPointerException, etc.) do **not** require a throws declaration. They will propagate automatically up the call stack if they are not caught.

**Final Answer:**

In your example:

* The display() method throws an unchecked exception (ArithmeticException), which does **not** require a throws declaration in display().
* The exception propagates to main(), but since main() doesn't catch it, the program terminates with the exception.
* package Exception;
* import java.io.IOException;
* public class Test3Exception {
* public static void main(String[] args) throws IOException {
* try {
* *readFile1*();
* } catch (Exception e) {
* System.***out***.println(e.getMessage());
* }
* }
* public static void readFile1() throws IOException {
* throw new IOException("jiiii");
* // checked exception - so must handle using either try catch block inside this methof or should throw exception using throws.
* //here we used throws in method declaration.
* }
* // public static void readFile1() {
* // throw new ArithmeticException(); // unchecked exception - directly exception gets thrown to caller method.
* // }
* }



A screenshot of a computer program

AI-generated content may be incorrect.