Mphasis
The Next Applied

Java Foundation – S1

Exceptions and Error Handling

Presentation By : V Sindhu

SME Java

Dos and Don’ts

- Login to GTW session on Time
- Login with your Mphasis email ID
- Use the question window for asking any queries

Day - 4

Exceptions and Error Handling

Agenda

- Why we use Exceptions?
- About Exceptions
- Why Exception Handling?
- Categories of Exceptions
- Differentiate among checked exceptions, unchecked exceptions and Errors
- Create a try-catch block and determine how exceptions alter normal program flow
- Create and invoke a method that throws an exception
- Exception Propagation
- Implement user-defined exceptions
- Try-With Resource Block

Objectives

After completion of this module, you should be able to:

- Explain exceptions.
- Define uses, types, and categories of exceptions.
- Identify the various types of exceptions in Java
- Implement exception handling using try, catch, throws, throw and finally clauses
- Describe exception propagation and implement user-defined exceptions
- Automatic Resource Management with Try-With Resource Block

Story

Scenario 1:

Think of Java as a child who visits the zoo. The happy path is when nothing goes wrong. The child continues to look at the animals until the program nicely ends. Nothing went wrong and there were no exceptions to deal with.

Story

Scenario 2:

This child's younger sister doesn't experience the happy path. In all the excitement she trips and falls. Luckily, it isn't a bad fall. The little girl gets up and proceeds to look at more animals. She has handled the issue all by herself. Unfortunately, she falls again later in the day and starts crying. This time, she has declared she needs help by crying. The story ends well. Her daddy rubs her knee and gives her a hug. Then they go back to seeing more animals and enjoy the rest of the day.

These are the two scenarios Java uses when dealing with exceptions. A method can handle the exception case itself or make it the caller's responsibility. You saw both in the trip to the zoo.

When Exception occur in a program?

- A program can fail for just about any reason. Here are just a few of the possibilities for program failure,
  - Your program tries to read a file that doesn't exist.
  - Your program tries to access a database, but the network connection to the database is unavailable.
  - You made a coding mistake and wrote an invalid SQL statement in your JDBC code.
  - You made a coding mistake and used the wrong format specifiers when using DateTimeFormatter.
- As you can see, some of these are coding mistakes. Others are completely beyond your control. Your program can't help it if the network connection goes down. What it can do is deal with the situation.

Exception as a Terminology

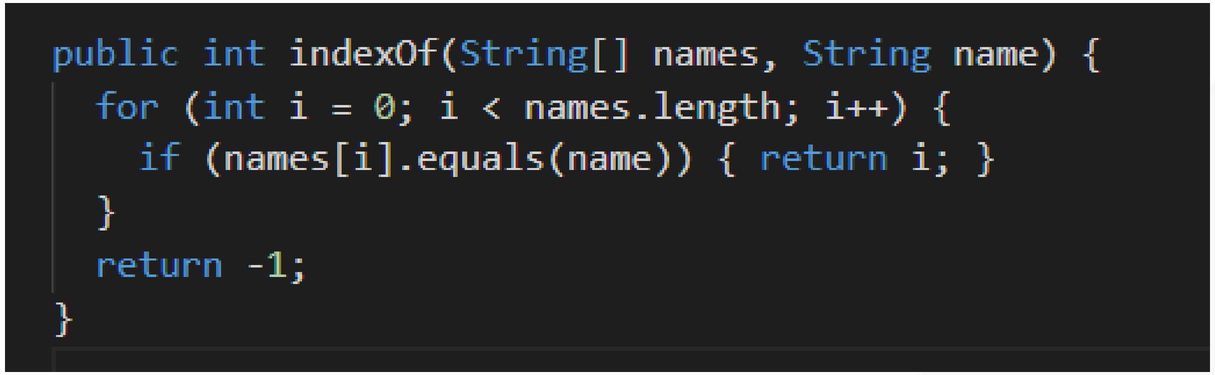
- An exception is Java's way of saying, "I give up. I don't know what to do right now. You deal with it." When you write a method, you can either deal with the exception or make it the calling code's problem.

- The happy path is when nothing goes wrong. With bad code, there might not be a happy path. For example, your code might have a bug where it always throws a **NullPointerException**. In this case, you have an exception path but not a happy path, because execution can never complete normally.

Real World Scenario: Return Codes vs. Exceptions

- Exceptions are used when “something goes wrong.” However, the word “wrong” is subjective.

- The following code returns −1 instead of throwing an exception if no match is found:



```java
public int indexOf(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) { return i; }
  }
  return -1;
}
```

- This approach is common when writing a method that does a search. For example, imagine being asked to find the name Joe in the array. It is perfectly reasonable that Joe might not appear in the array. When this happens, a special value is returned. An exception should be reserved for exceptional conditions like names being null.

Real World Scenario: Return Codes vs. Exceptions

- In general, try to avoid return codes. Return codes are commonly used in searches, so programmers are expecting them.

- An exception forces the program to deal with them or end with the exception if left unhandled, whereas a return code could be accidentally ignored and cause problems later in the program.

- An exception is like shouting, “Deal with me!”

What is exception?

- An exception can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions

- Uses of Exceptions are:
    - Consistent error reporting style.
    - Easy to pass errors up the stack.
    - Pinpoint errors better in stack trace.
        - As long as you “fail fast” and throw as soon as you find a problem.

What is exception?

- Wrap useful information up in your exception classes, use that information later when handling the exception.

- Exceptions don’t always have to be errors, maybe your program can cope with the exception and keep going!

- Separating error handling code from Regular code.

Types of errors

- Compile time errors
- Runtime errors

Exception occurrence reason

- Running out of memory
- Resource allocation errors
- Inability to find files
- Problem in network connectivity

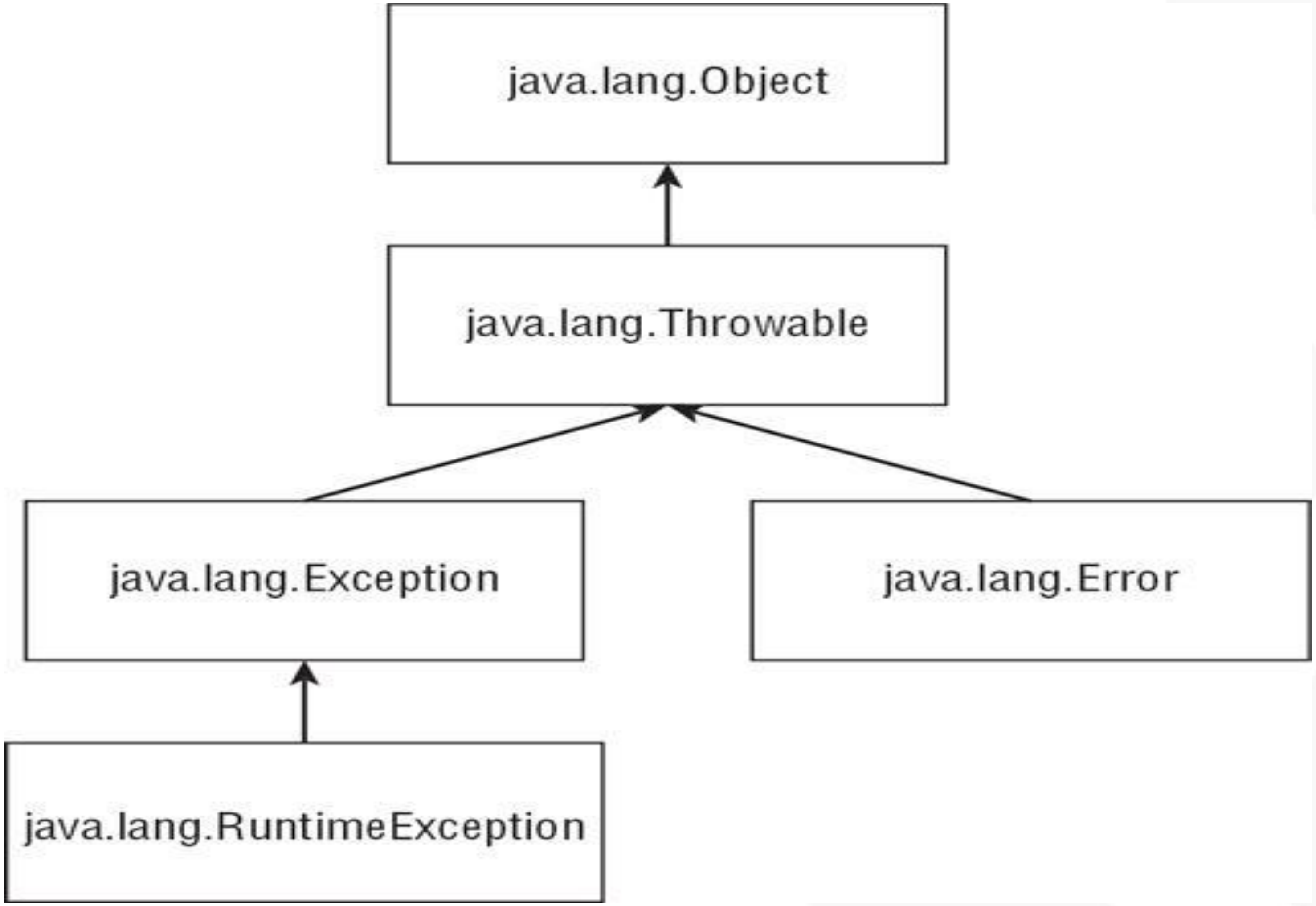
Exception occurrence levels

- **Hardware/operating system level.**
  - Arithmetic exceptions; divide by 0, under/overflow.
  - Memory access violations; segment fault, stack over/underflow.

- **Language level.**
  - Type conversion; illegal values, improper casts.
  - Bounds violations; illegal array indices.
  - Bad references; null pointers.

- **Program level.**
  - User defined exceptions.

Java exception hierarchy



Understanding Exception Types

- **Error:** Error means something went so horribly wrong that your program should not attempt to recover from it. For example, the disk drive “disappeared.” These are abnormal conditions that you aren't likely to encounter. Examples: ExceptionInInitializerError, StackOverflowError, NoClassDefFoundError

- **RuntimeException:** A runtime exception is defined as the RuntimeException class and its subclasses. Runtime exceptions tend to be unexpected but not necessarily fatal. For example, accessing an invalid array index is unexpected. Runtime exceptions are also known as **unchecked exceptions**.

- **Exception:** A **checked exception** includes Exception and all subclasses that do not extend RuntimeException. Checked exceptions tend to be more anticipated—for example, trying to read a file that doesn't exist. For checked exceptions, Java requires the code to either handle them or declare them in the method signature.

Categories of exceptions

- **Built in Exceptions**
  - Checked Exceptions
  - Unchecked Exceptions

- **User defined Exceptions**

Types of exceptions			
Type	**How to recognize**	**Recommended for program to catch?**	**Is program required to catch or declare?**
Unchecked Exceptions	RuntimeException or its subclasses	Yes	No
Checked exception	Exception or its subclasses but not RuntimeException or its subclasses	Yes	Yes
Error	Error or its subclasses	No	No

Unchecked exceptions

- Raised implicitly by system because of illegal execution of program.
- Do not need to announce the possibility of exception occurrence.
- When unchecked exception occurred, if programmer does not deal with it, it would be processed by default exception handler.
- Throw-able is base class for Exception and Error class.

Unchecked exceptions Examples

- **ArithmeticException** Thrown by the JVM when code attempts to divide by zero.
- **ArrayIndexOutOfBoundsException** Thrown by the JVM when code uses an illegal index to access an array.
- **ClassCastException** Thrown by the JVM when an attempt is made to cast an object to a subclass of which it is not an instance.
- **IllegalArgumentException** Thrown by the program to indicate that a method has been passed an illegal or inappropriate argument.
- **NullPointerException** Thrown by the JVM when there is a null reference where an object is required.
- **NumberFormatException** Thrown by the program when an attempt is made to convert a string to a numeric type, but the string doesn't have an appropriate format.

Checked exceptions

- Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed.

- The checked exceptions that a method may raise are part of the method's signature.

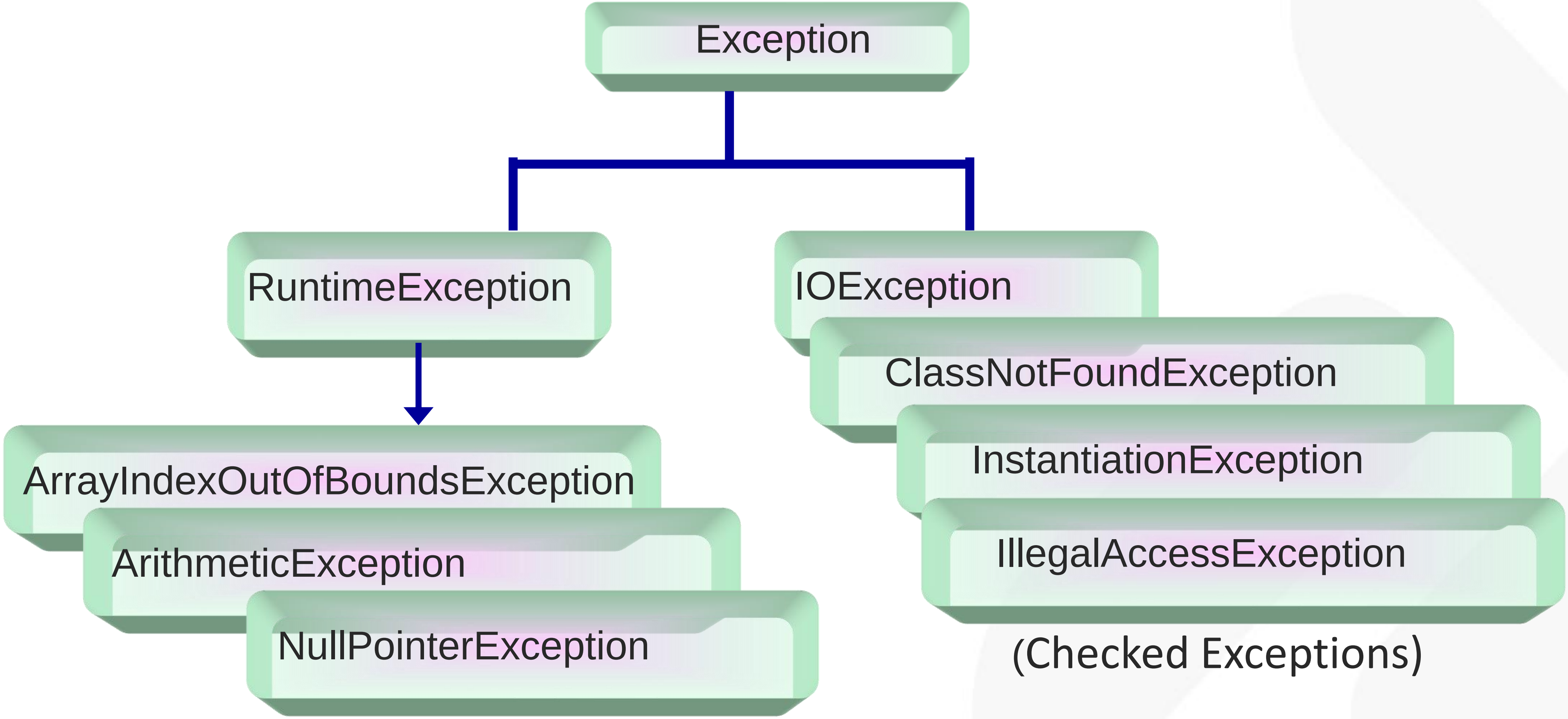
- Every method that throws a checked exception must advertise it in the throws clause in its method definition.

- Every method that calls a method that advertises a checked exception must either handle that exception (with try and catch) or must in turn advertise that exception in its own throws clause.

Checked exceptions Examples

Exception	Used when	Checked or unchecked?
java.text.ParseException	Converting a String to a number.	Checked
java.io.IOException java.io.FileNotFound Exception java.io.NotSerializable Exception	Dealing with IO issues. IOException is the parent class. There are a number of subclasses. You can assume any java.io exception is checked.	Checked
java.sql.SQLException	Dealing with database issues. SQLException is the parent class. Again, you can assume any java.sql exception is checked.	Checked

Java exception hierarchy (continued)



Exception

RuntimeException

IOException

ArrayIndexOutOfBoundsException

ArithmeticException

NullPointerException

ClassNotFoundException

InstantiationException

IllegalAccessException

(Checked Exceptions)

(Unchecked Exceptions)

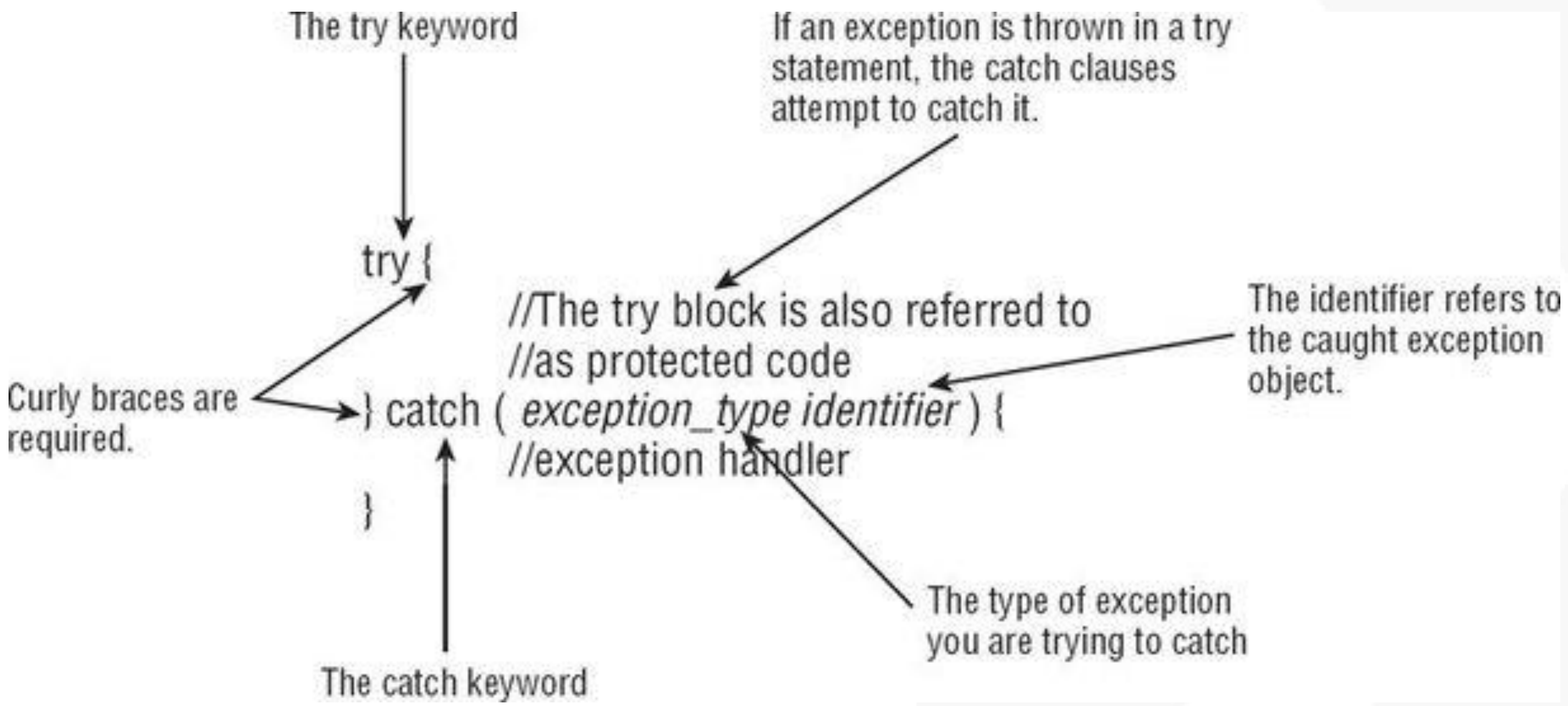
Implementing exception handling

- try
- catch
- finally
- throw
- throws
- try with resource block

Implementing exception handling

- Using try and catch statements
  - The try block encloses the statements that might raise an exception within it and defines the scope of the exception handlers associated with it.
  - The catch block is used as an exception-handler. You enclose the code that you want to monitor inside a try block to handle a run time error.

Implementing exception handling with try catch : Syntax



```
try {
        //The try block is also referred to
        //as protected code
} catch ( exception_type identifier ) {
        //exception handler
}
```

The try keyword

If an exception is thrown in a try statement, the catch clauses attempt to catch it.

Curly braces are required.

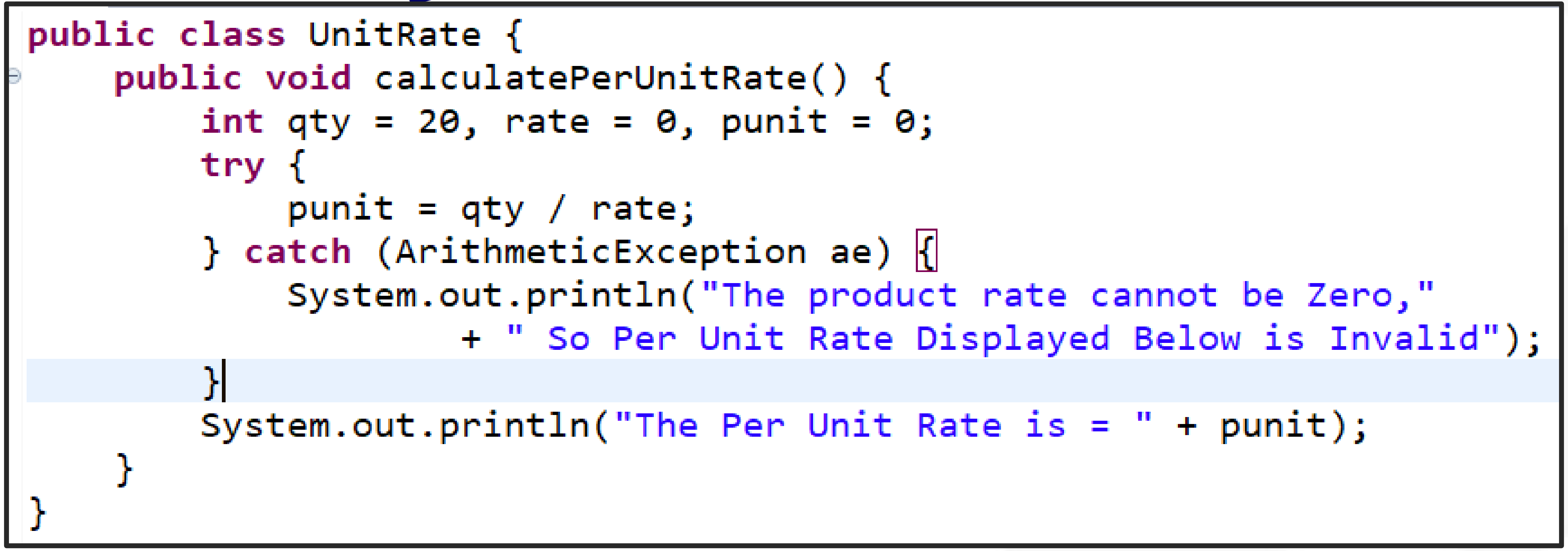
The identifier refers to the caught exception object.

The type of exception you are trying to catch

The catch keyword

Implementing exception handling continued

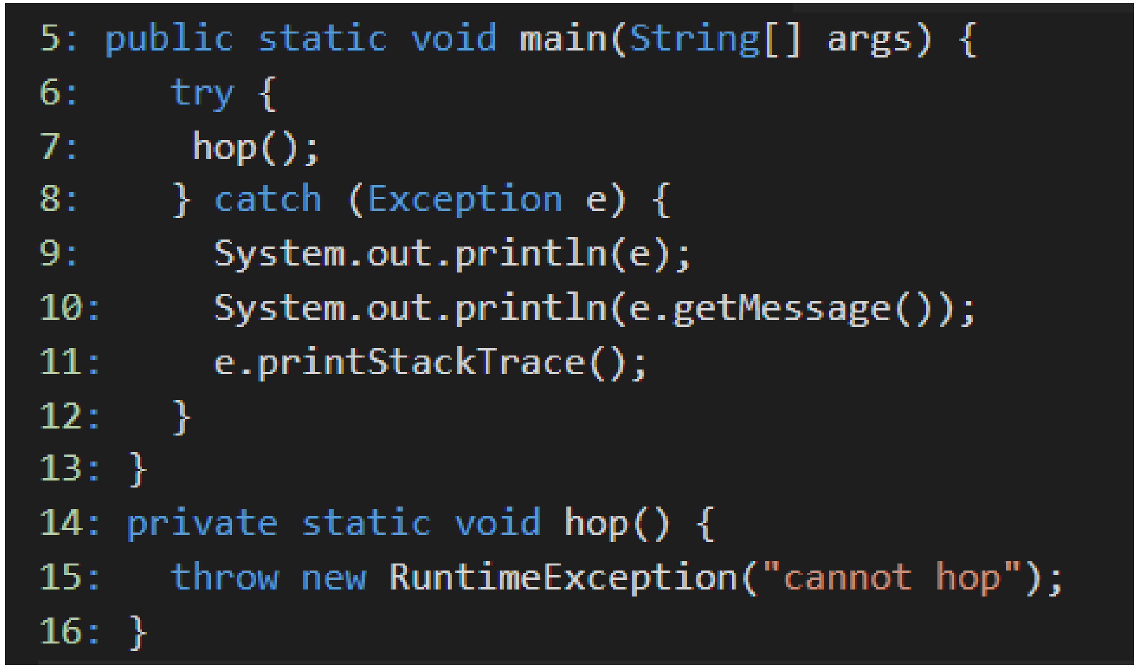
Arithmetic error occurred and passed to exception handler i.e.



```
public class UnitRate {
    public void calculatePerUnitRate() {
        int qty = 20, rate = 0, punit = 0;
        try {
            punit = qty / rate;
        } catch (ArithmeticException ae) {
            System.out.println("The product rate cannot be Zero,"
                    + " So Per Unit Rate Displayed Below is Invalid");
        }
        System.out.println("The Per Unit Rate is = " + punit);
    }
}
```

Printing an Exception

- There are three ways to print an exception. You can let Java print it out, print just the message, or print where the stack trace comes from. This example shows all three approaches:



```
5: public static void main(String[] args) {
6:    try {
7:     hop();
8:    } catch (Exception e) {
9:      System.out.println(e);
10:     System.out.println(e.getMessage());
11:     e.printStackTrace();
12:   }
13: }
14: private static void hop() {
15:   throw new RuntimeException("cannot hop");
16: }
```

This code results in the following output:

```
java.lang.RuntimeException: cannot hop

cannot hop

java.lang.RuntimeException: cannot hop
at trycatch.Handling.hop(Handling.java:15)
at trycatch.Handling.main(Handling.java:7)
```

Using multiple catch statement/blocks

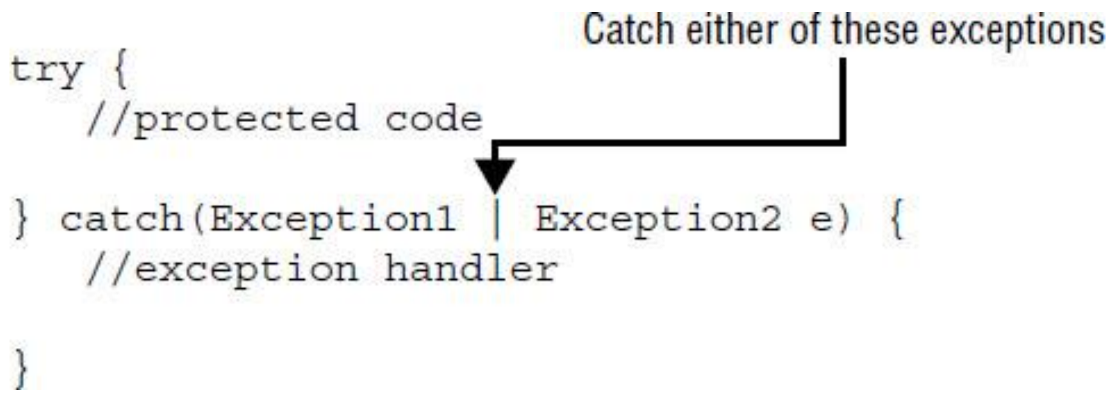
- A single try block can have zero, one or many catch blocks.

- The multiple catch blocks generate unreachable code error.

- If the first catch block contains the Exception class object, then the subsequent catch blocks are never executed. If this happen it is known as unreachable code problem.

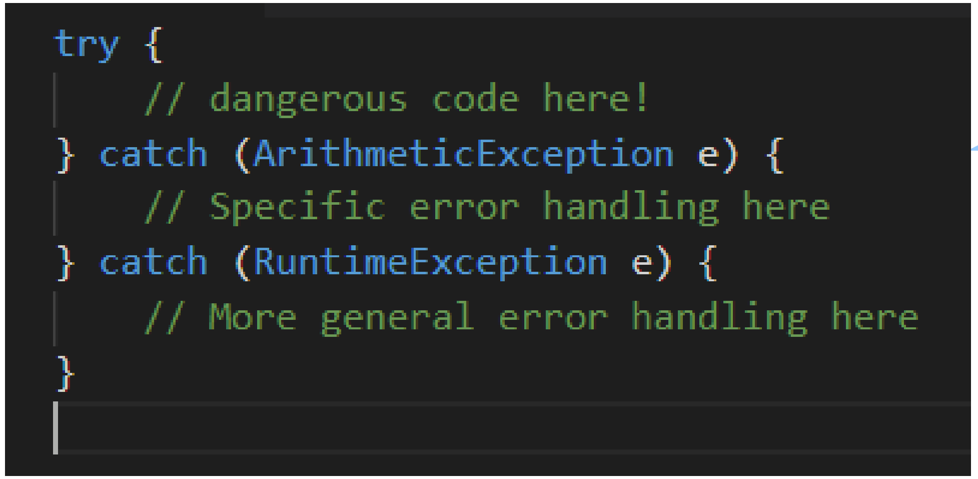
- To avoid unreachable code error, the last catch block in multiple catch blocks must contain the Exception class object.

Example: Using multiple catch statement/blocks



Catch either of these exceptions

```
try {
   //protected code

} catch(Exception1 | Exception2 e) {
   //exception handler

}
```

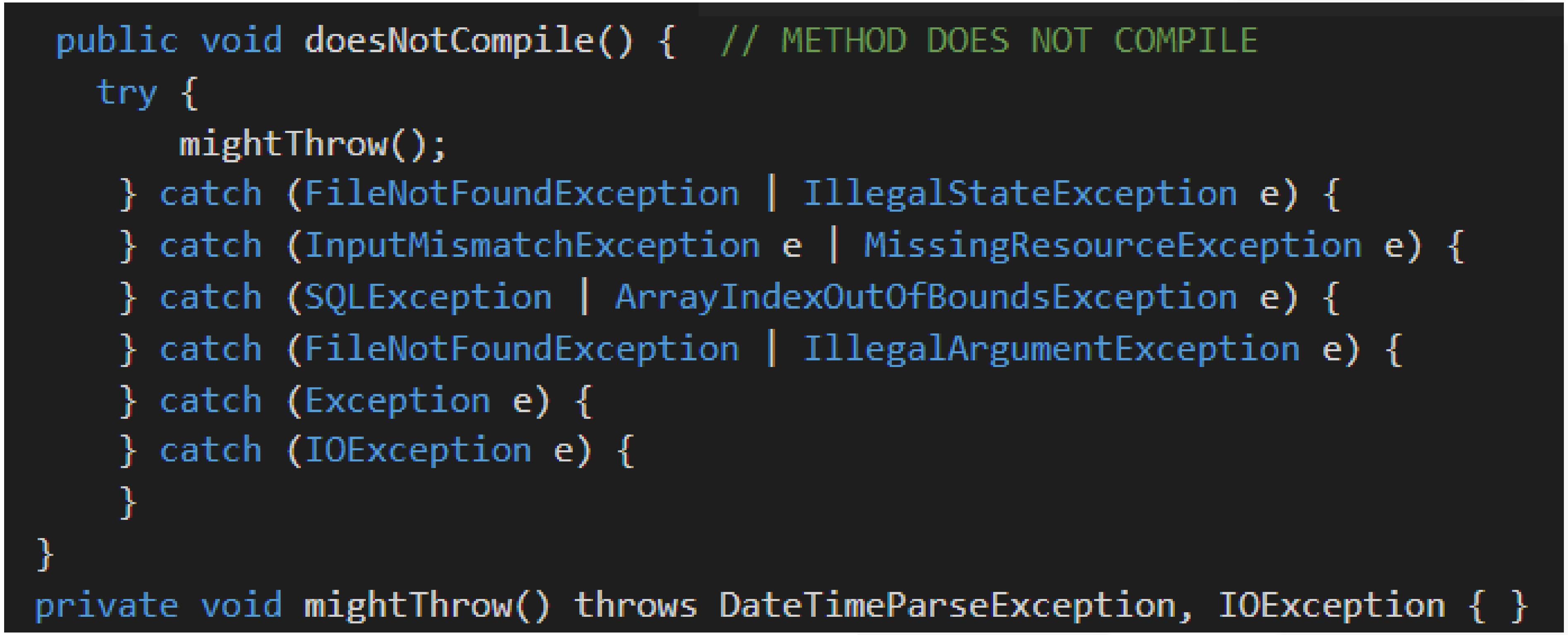


```
try {
    // dangerous code here!
} catch (ArithmeticException e) {
    // Specific error handling here
} catch (RuntimeException e) {
    // More general error handling here
}
```

Order Matters!
Less general
first!

Question on multiple catch statement/blocks

How many errors you can find in this try statement ?



```
 public void doesNotCompile() {  // METHOD DOES NOT COMPILE
   try {
       mightThrow();
    } catch (FileNotFoundException | IllegalStateException e) {
    } catch (InputMismatchException e | MissingResourceException e) {
    } catch (SQLException | ArrayIndexOutOfBoundsException e) {
    } catch (FileNotFoundException | IllegalArgumentException e) {
    } catch (Exception e) {
    } catch (IOException e) {
    }
}
private void mightThrow() throws DateTimeParseException, IOException { }
```

Answer of multiple catch statement/blocks

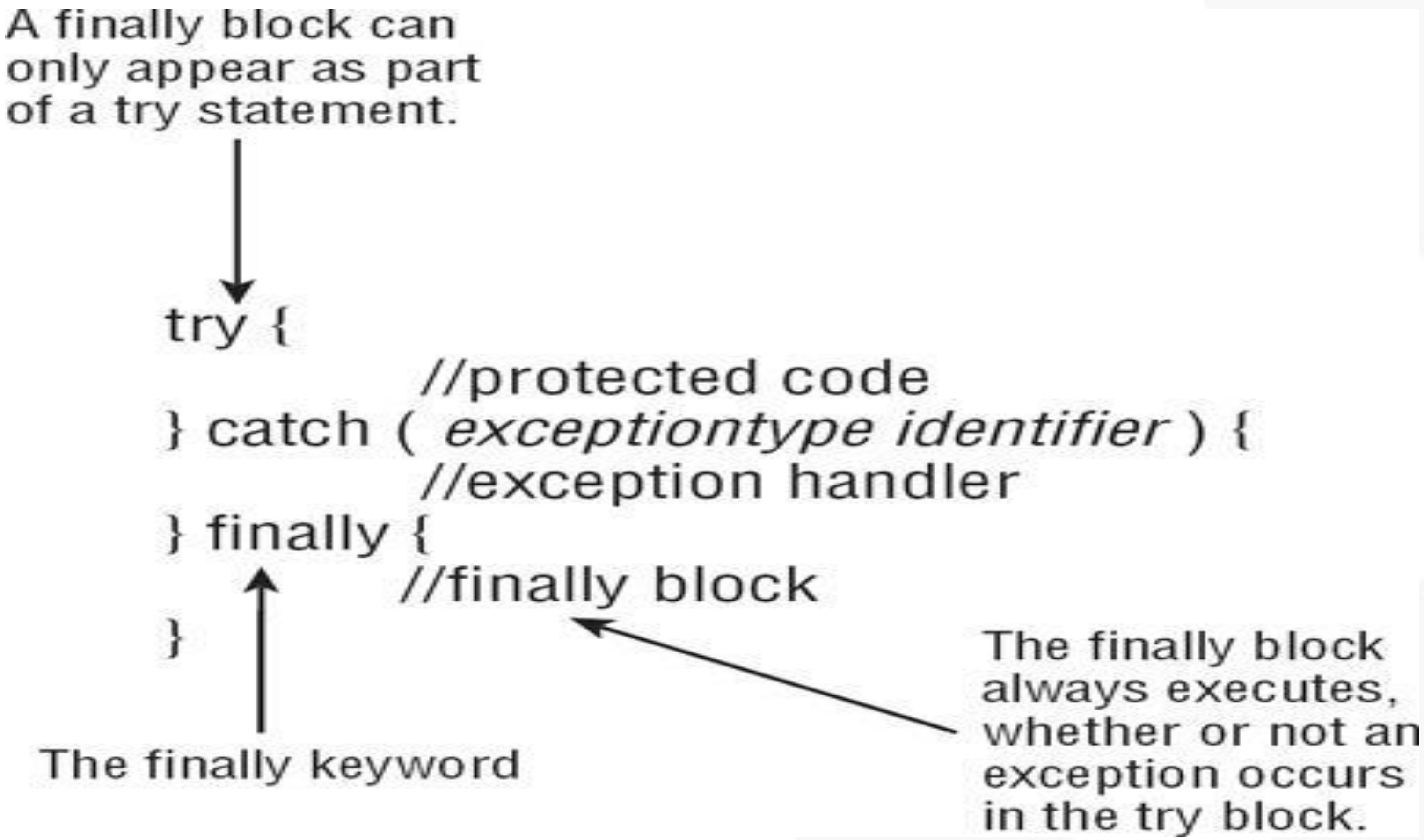
Line 15 has an extra variable name. Remember that there can be only one exception variable per catch block.

Line 18 and 19 are reversed. The more general superclasses must be caught after their subclasses. While this doesn't have anything to do with multi-catch, you'll see "regular" catch block problems mixed in with multi-catch.

Line 17 cannot catch FileNotFoundException because that exception was already caught on line 15. You can't list the same exception type more than once in the same try statement, just like with "regular" catch blocks.

Line 16 cannot catch SQLException because nothing in the try statement can potentially throw one. Again, just like "regular" catch blocks, any runtime exception may be caught. However, only checked exceptions that have the potential to be thrown are allowed to be caught.

Using finally clause



A finally block can only appear as part of a try statement.

```
try {
              //protected code
} catch ( exceptiontype identifier ) {
              //exception handler
} finally {
             //finally block
}
```

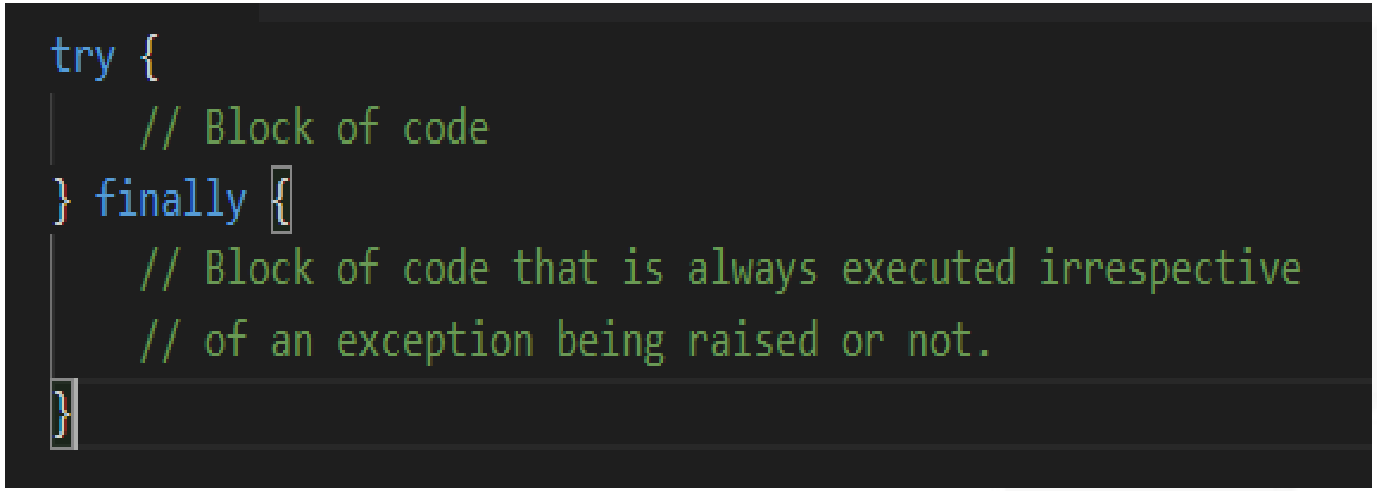
The finally keyword

The finally block always executes, whether or not an exception occurs in the try block.

Using finally clause

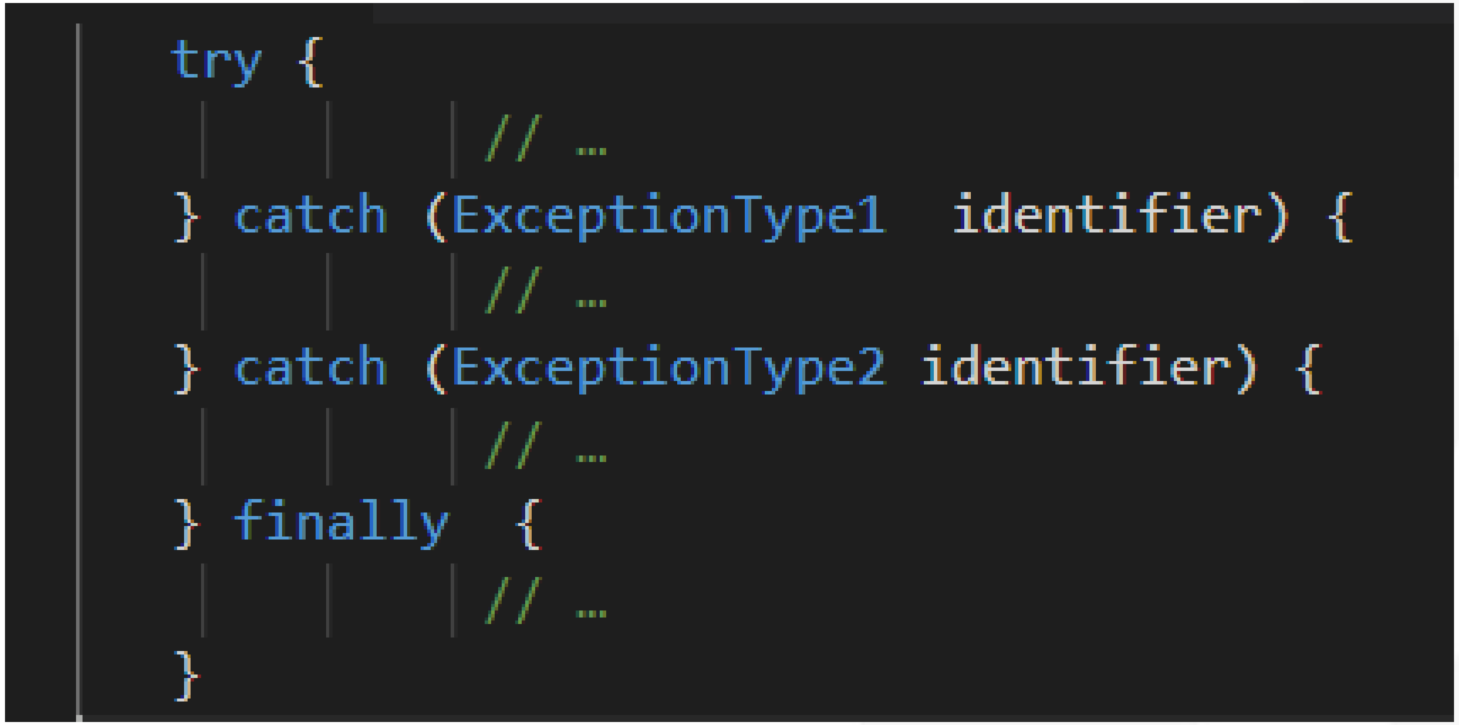
The finally block is used to process certain statements, no matter whether an exception is raised or not.

Syntax of finally block



```
try {
    // Block of code
} finally {
    // Block of code that is always executed irrespective
    // of an exception being raised or not.
}
```

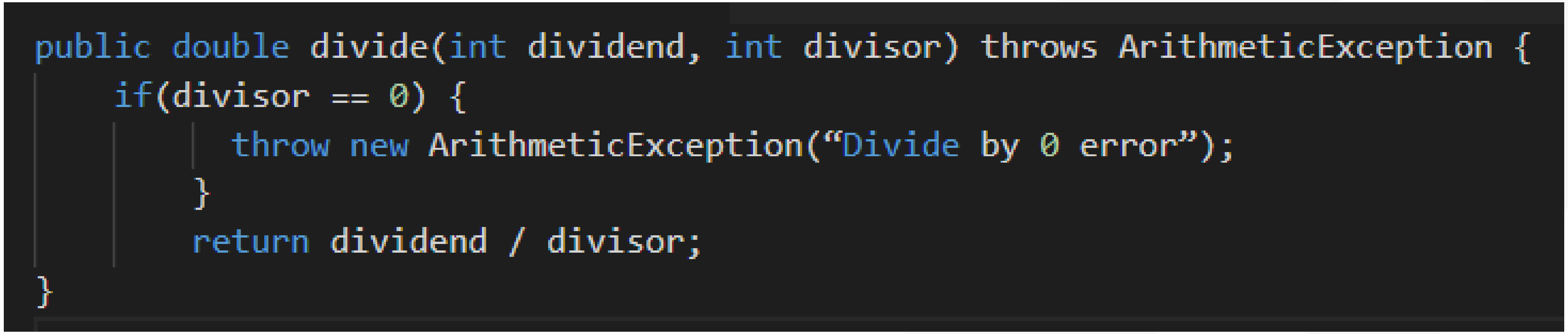
try-catch-finally statement



```
try {
        // …
} catch (ExceptionType1  identifier) {
        // …
} catch (ExceptionType2 identifier) {
        // …
} finally  {
        // …
}
```

Throwing an exception

- The throw statement causes termination of the normal flow of control of the Java code and stops the execution of the subsequent statements if an exception is thrown when the throw statement is executed.
- The throw clause transfers the control to the nearest catch block handling the type of exception object throws.
- The following syntax shows how to declare the throw statement:
  - throw ThrowableObj



```
public double divide(int dividend, int divisor) throws ArithmeticException {
    if(divisor == 0) {
        throw new ArithmeticException(“Divide by 0 error”);
    }
    return dividend / divisor;
}
```

Using throws clause

- The throws statement is used by a method to specify the types of exceptions the method throws.

- If a method can raise an exception that it does not handle, the method must specify that the exception must be handled by the calling method.

- This is done using the throws statement.

```
[modifiers]  returntype  methodName(params)  throws e1, ... ,en { }
```

```
void readData() throws IOException
```

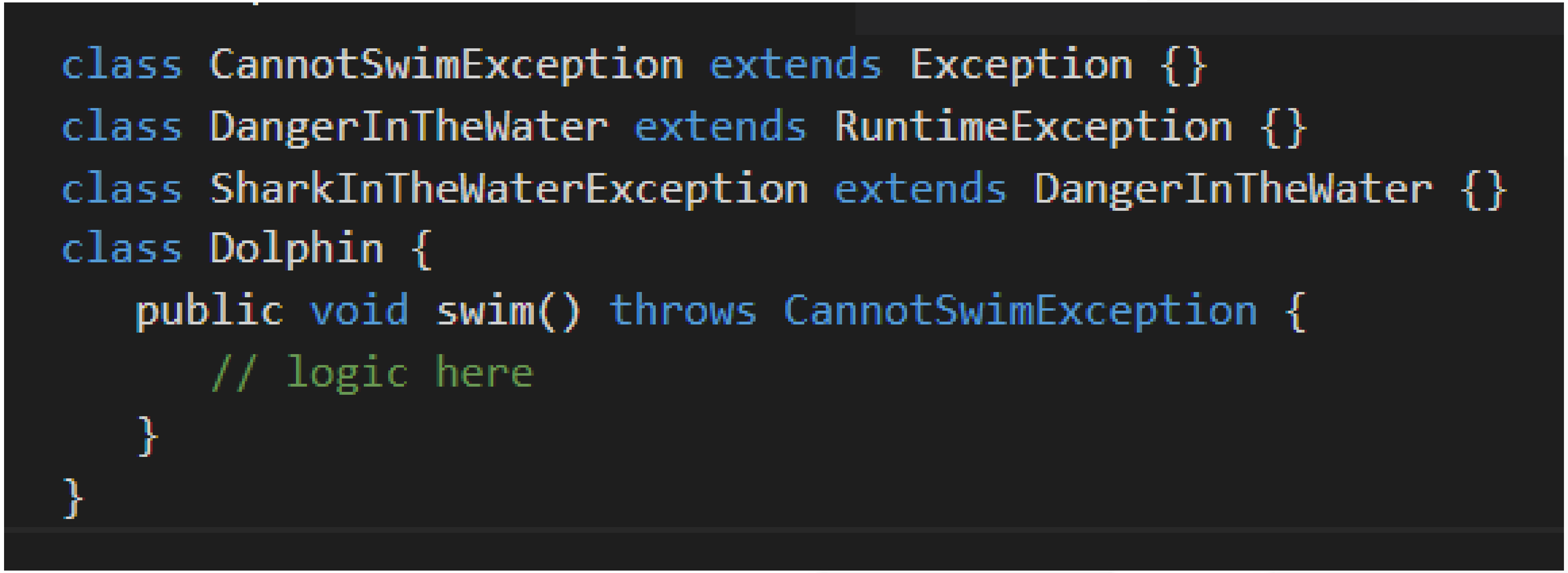
Custom exceptions: Creating your own exceptions

- Java provides many exception classes out of the box. Sometimes, you want to write a method with a more specialized type of exception. You can create your own exception class to do this.

- When creating your own exception, you need to decide whether it should be a checked or unchecked exception. While you can extend any exception class, it is most common to extend Exception (for checked) or RuntimeException (for unchecked.)

- Creating your own exception class is easy. Can you figure out whether the exceptions are checked or unchecked in this example?

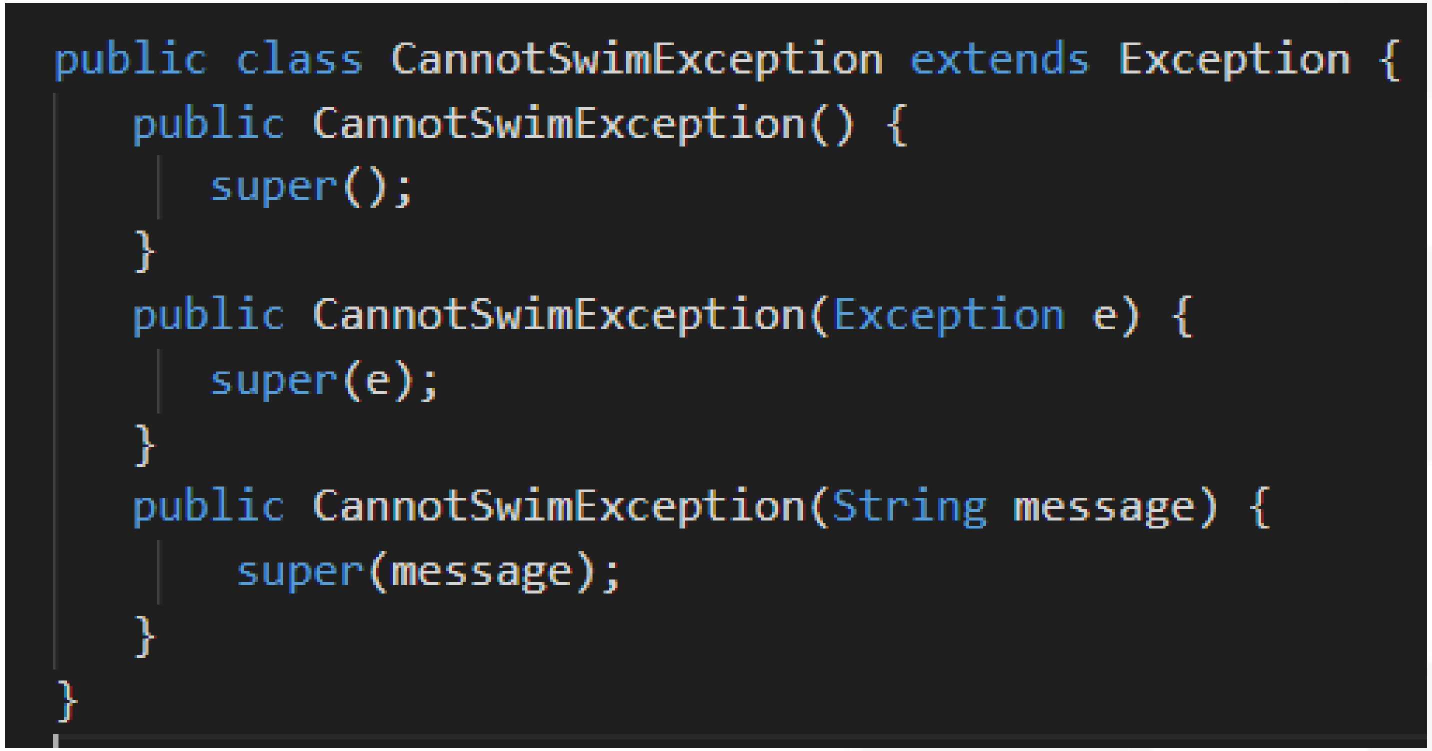
Custom exceptions: Creating your own exceptions



```
class CannotSwimException extends Exception {}
class DangerInTheWater extends RuntimeException {}
class SharkInTheWaterException extends DangerInTheWater {}
class Dolphin {
   public void swim() throws CannotSwimException {
      // logic here
   }
}
```

Custom exceptions: Common constructors defined by the Exception class

- The following example shows the three most common constructors defined by the Exception class:



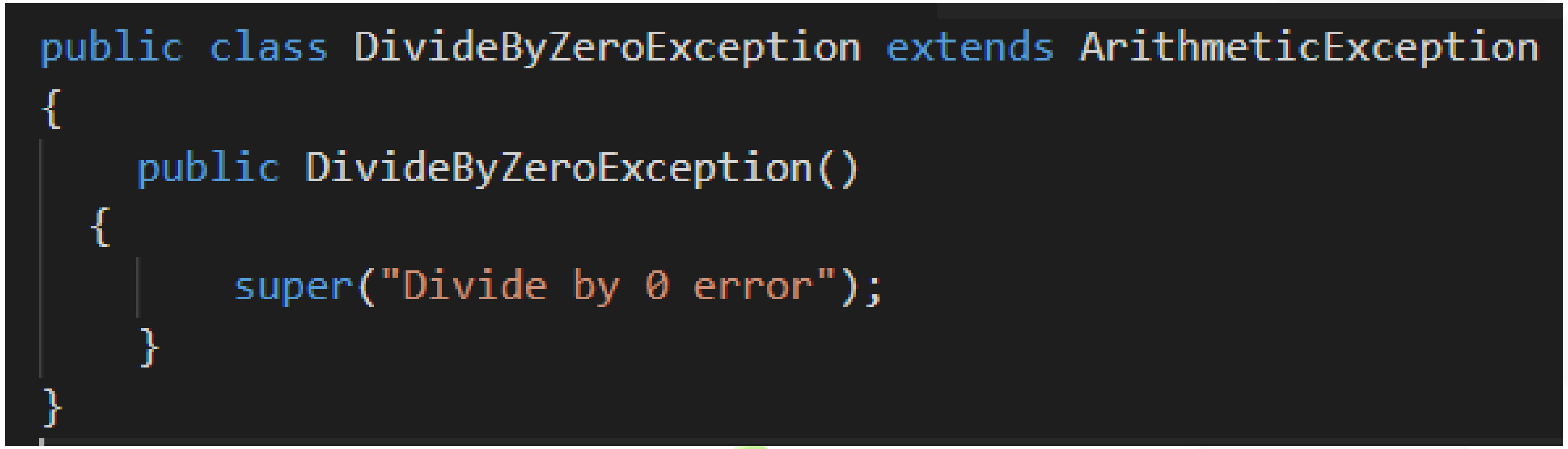
```
public class CannotSwimException extends Exception {
    public CannotSwimException() {
        super();
    }
    public CannotSwimException(Exception e) {
        super(e);
    }
    public CannotSwimException(String message) {
         super(message);
    }
}
```

User defined exceptions: Creating your own exceptions

- Defining your own exceptions let you handle specific exceptions that are tailor-made for your application.

- Steps to Create a User Defined Exception
  - Create a class that extend from a right kind of class from the Exception Hierarchy.
  - Let’s make a DivideByZeroException for use by our UnitRate class

User defined exceptions (continued)



```java
public class DivideByZeroException extends ArithmeticException
{
    public DivideByZeroException()
  {
        super("Divide by 0 error");
    }
}
```

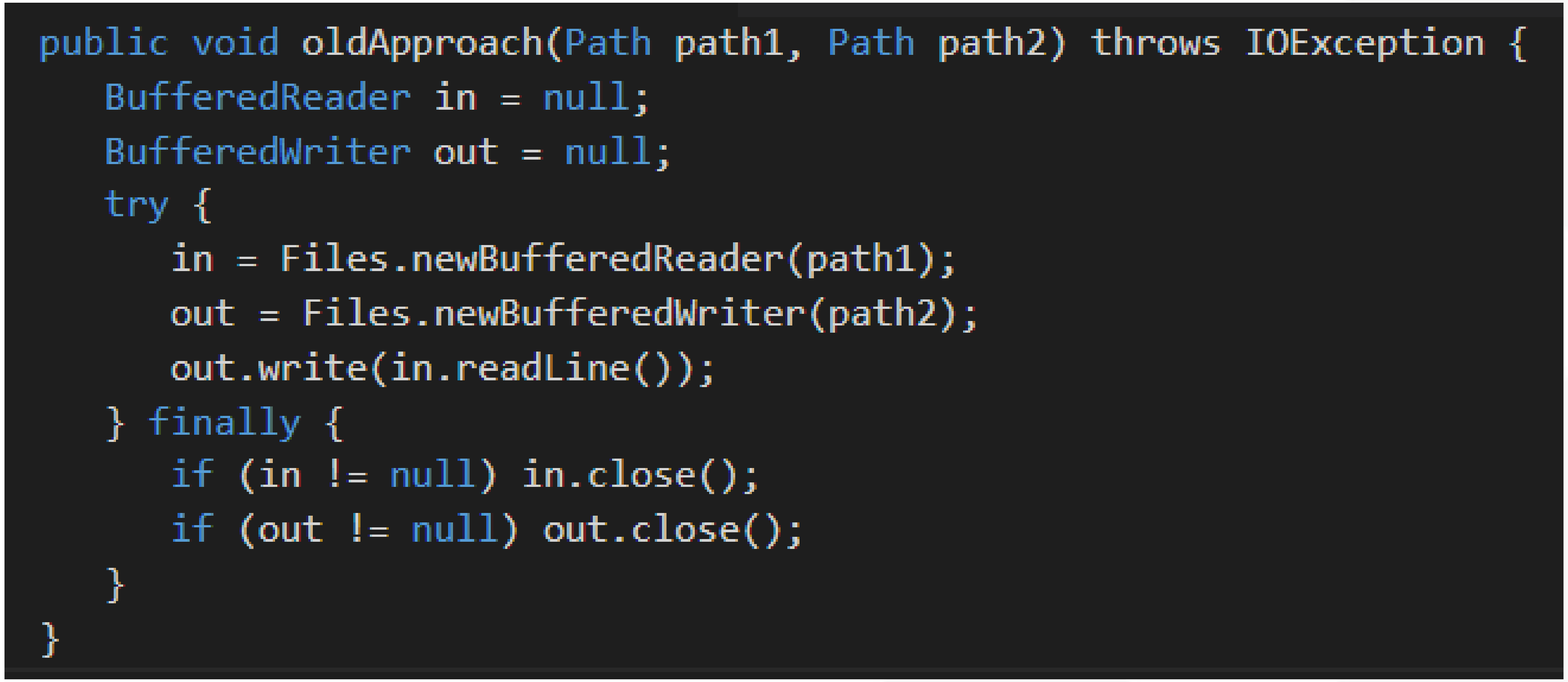
Here we extended ArithmeticException.

Using Try-With-Resources

- Multi-catch allows you to write code without duplication.
- Another problem arises with duplication in finally blocks.
- It is important to close resources when you are finished with them.

Using Try-With-Resources

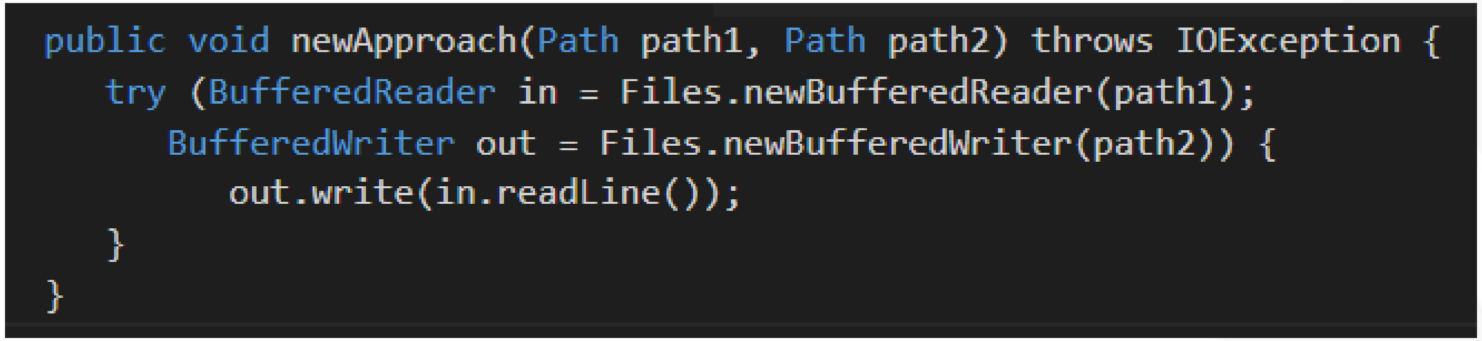
- Imagine that you want to write a simple method to read the first line of one file and write it to another file. Prior to Java 7, your code would look like the following.



```java
public void oldApproach(Path path1, Path path2) throws IOException {
   BufferedReader in = null;
   BufferedWriter out = null;
   try {
      in = Files.newBufferedReader(path1);
      out = Files.newBufferedWriter(path2);
      out.write(in.readLine());
   } finally {
      if (in != null) in.close();
      if (out != null) out.close();
   }
}
```

Using Try-With-Resources

- Switching to the try-with-resources syntax introduced in Java 7, it can be rewritten as follows:



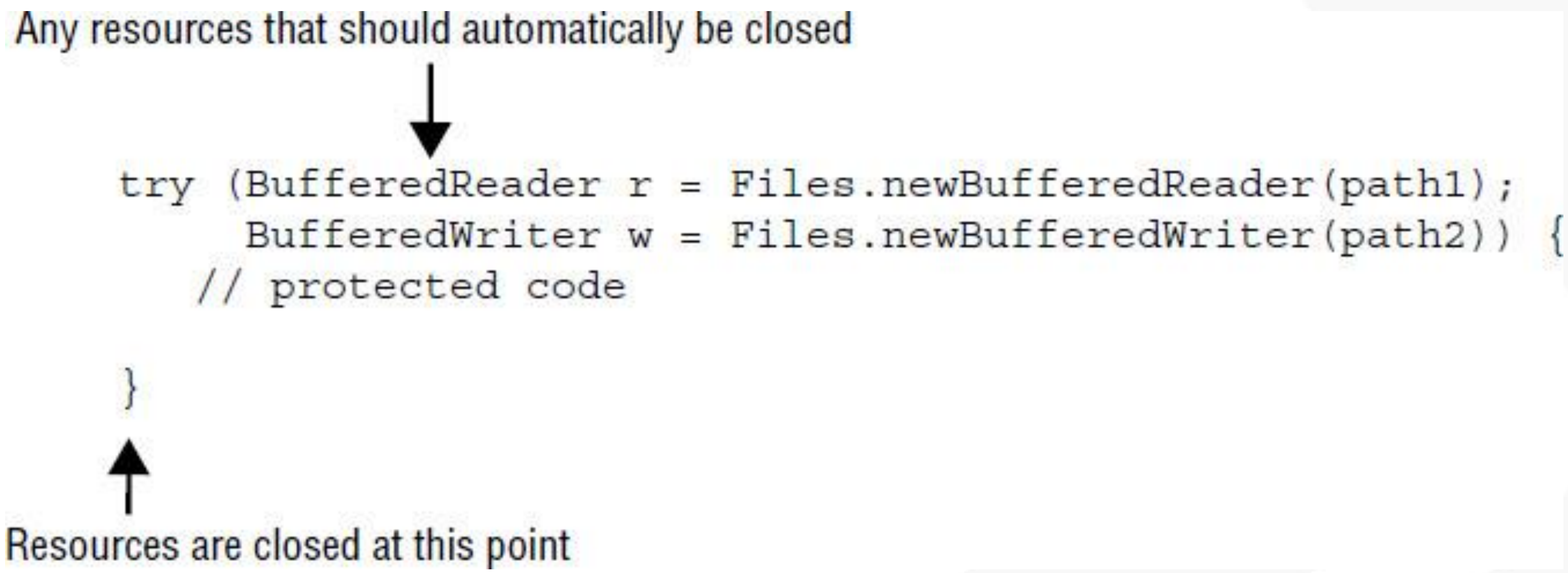
```
public void newApproach(Path path1, Path path2) throws IOException {
    try (BufferedReader in = Files.newBufferedReader(path1);
        BufferedWriter out = Files.newBufferedWriter(path2)) {
            out.write(in.readLine());
    }
}
```

The new version has half as many lines! There is no longer code just to close resources.

The new try-with-resources statement automatically closes all resources opened in the try clause.

This feature is also known as automatic resource management, because Java automatically takes care of the closing.

Basic Syntax of Try-With-Resources



Any resources that should automatically be closed

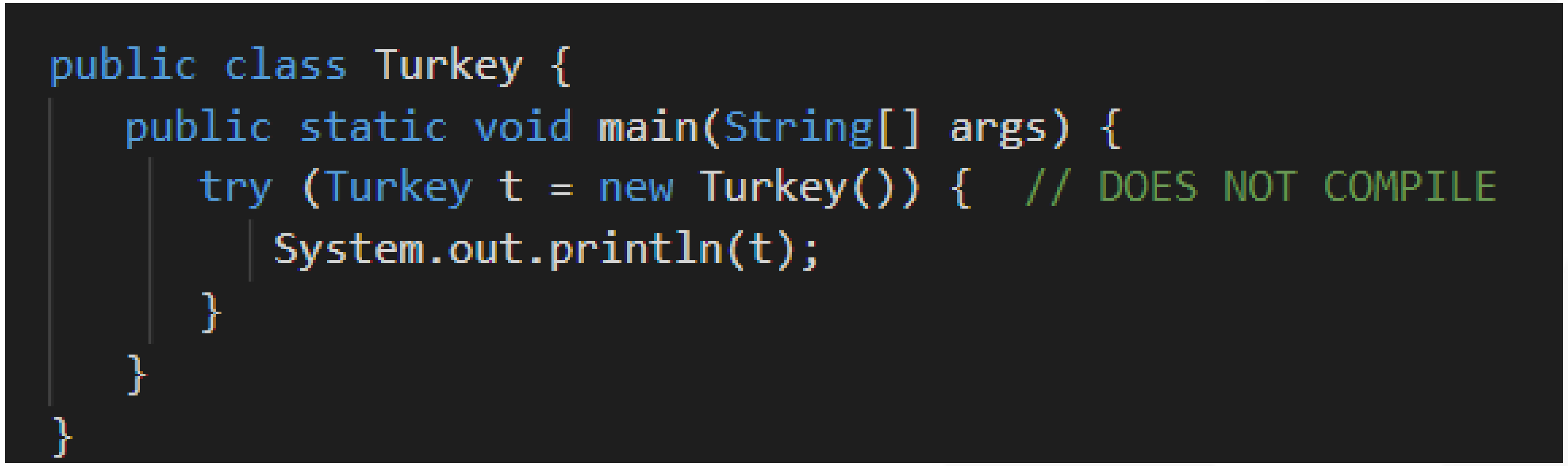
```
try (BufferedReader r = Files.newBufferedReader(path1);
     BufferedWriter w = Files.newBufferedWriter(path2)) {
   // protected code

}
```

Resources are closed at this point

AutoCloseable

- You can't just put any random class in a try-with-resources statement.

- Java commits to closing automatically any resources opened in the try clause. Here we tell Java to try to close the Turkey class when we are finished with it:
- Java doesn't allow this. It has no idea how to close a Turkey. Java informs us of this fact with a compiler error:

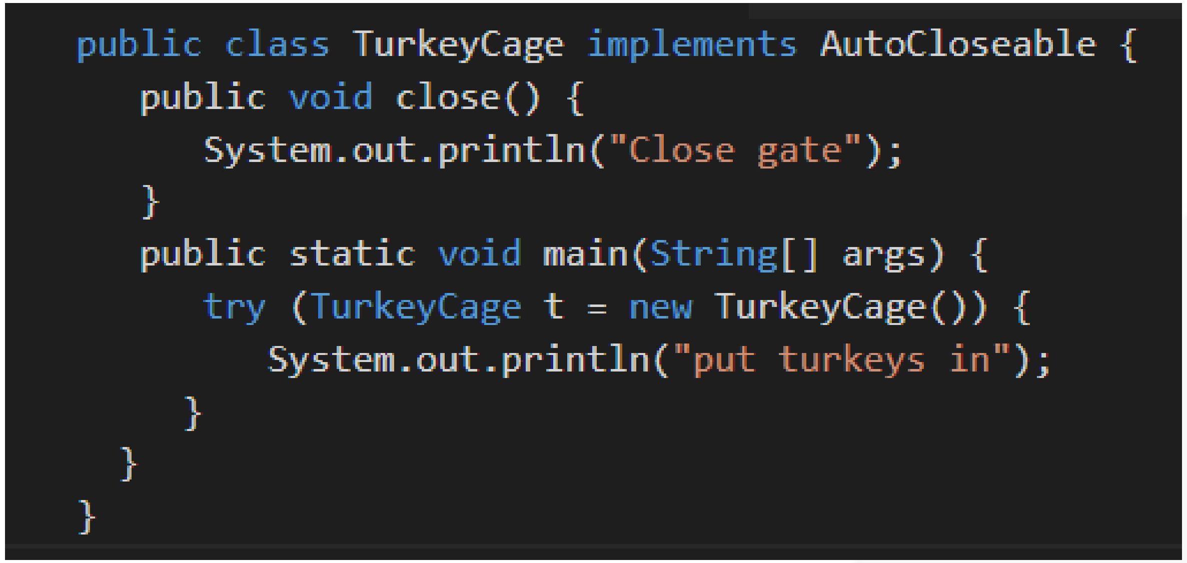


```java
public class Turkey {
   public static void main(String[] args) {
      try (Turkey t = new Turkey()) {  // DOES NOT COMPILE
         System.out.println(t);
      }
   }
}
```

- The resource type Turkey does not implement java.lang.AutoCloseable.

AutoCloseable

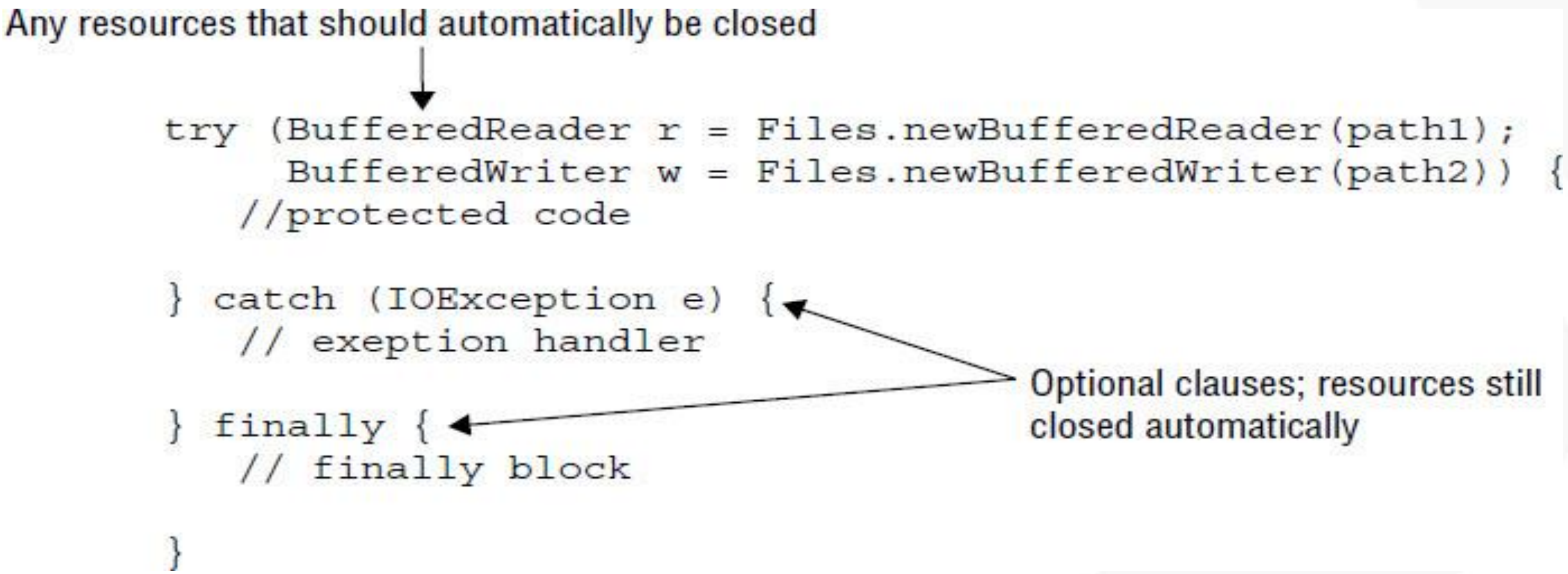
- For a class to be created in the try clause, Java requires it to implement an interface called **AutoCloseable**. TurkeyCage does implement this interface:



```
public class TurkeyCage implements AutoCloseable {
   public void close() {
      System.out.println("Close gate");
   }
   public static void main(String[] args) {
      try (TurkeyCage t = new TurkeyCage()) {
         System.out.println("put turkeys in");
      }
   }
}
```

- The AutoCloseable interface has only one method to implement:
      public void close() throws Exception;

The syntax of try-with-resources including catch/finally



Any resources that should automatically be closed

```
try (BufferedReader r = Files.newBufferedReader(path1);
     BufferedWriter w = Files.newBufferedWriter(path2)) {
   //protected code

} catch (IOException e) {
   // exeption handler

} finally {
   // finally block

}
```

Optional clauses; resources still closed automatically

Review Questions

1. The two subclasses of the Throwable class are:
   - [✓ a. Exception class and Error class]
   - b. Exception class and Object class
   - c. Error class and RunTimeException class
   - d. Exception class and RunTimeException class

2. The Throwable class is the sub class of ________ class.
   - a. Exception
   - b. Error
   - [✓ c. Object]
   - d. RunTimeException

Review Questions

3. Which exception occurs when you try to create an object of an abstract class or interface?

   a. ClassNotFoundException

   b. IllegalAccessException

   [✓ c. InstantiationException]

   d. NoSuchMethodException

Review questions (continued)

4. Consider the statements:
   Statement A: The scope of the catch block is restricted to the statements in the preceding try block only.
   Statement B: A try block must have at least one catch block that follows it immediately.

   Which of the following options is true?

   a. Statement A is true and statement B is false
   b. Statement A is false and statement B is true
   c. Both, statements A and B, are true
   [✓ d. Both, statements A and B, are false]

Recap

- Errors can be broadly categorized into two groups based on whether the compiler is able to handle the error or not, such as compile time errors and run time errors.
- An exception is a run time error that can be defined as an abnormal event that occurs during the execution of a program and disrupts the normal flow of instructions.
- In Java, the Throwable class is the superclass of all the exception classes. It is the class at the top position in the exception class hierarchy. The Exception class and the Error class are two direct subclasses of the Throwable class.
- The built-in exceptions in Java are divided into two types on the basis of the conditions where the exception is raised:
  - Checked Exceptions or Compiler-enforced Exceptions
  - Unchecked exceptions or Runtime Exceptions

Recap

- You can handle exception using
  - try, catch, throws, throw, finally, try with resource block.
- You use multiple catch blocks to throw more than one type of exception.
- The finally clause is used to execute the statements that need to be executed whether an exception has been thrown.
- The throw statement causes termination of the normal flow of control of the Java code and stops the execution of subsequent statements after the throw statement.
- The throws clause is used by a method to specify the types of exceptions the method throws.
- You can create your own exception classes to handle the situations specific to an application.

Mphasis

The Next Applied

THANK YOU

Important Confidentiality Notice