Object-Oriented Programming Using Java

Java Exception Handling

Why Exception Handling?

- **Dictionary Meaning:** Exception is an abnormal condition
- A variety of errors can occur when a program is running -- For example:
  - **(Real) User input error.** Example: bad input or URL
  - **Device errors.** Example: Printer Not Connected
  - **Physical limitations.** Example: disk full
  - **Code errors.** Example: interact with code that does not fulfill its contact
  - **Loss of network connection.**
  - **Opening an unavailable file.**
- When a program runs into a runtime error, the program terminates **abnormally**
- **Desirable:** when an error occurs
  - *Return to safe state, save work, exit gracefully*
- Can we handle the runtime error so that the program can continue to run or terminate gracefully?

Why Exception Handling?

- Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved
- When an exception occurs, the program usually *terminates abruptly*
- The major benefit of Exception handling is that it maintains the normal flow of the application despite the occurrence of an exception
- An exception handler makes sure that all the statements in the program are executed normally and the program flow does not break abruptly

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

These statements will not be executed

Exception versus Error

- An *exception* represents an abnormal condition; an *error*, on the other hand, represents some irrecoverable condition

- Errors are usually beyond the control of the programmer, and we should not try to handle errors
- **Example:** JVM running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc...

- As a matter of fact:
  - An *error* indicates a serious problem that a reasonable application should not try to handle on its own
  - An *exception* indicates conditions that a reasonable application might try to catch and handle modestly

Example Scenario 1: Runtime Error

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: "); // Prompt the user to enter two integers
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
    System.out.println("Execution continues ...");
  }
}
```

Example Scenario 1: Runtime Error

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: "); // Prompt the user to enter two integers
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
    System.out.println("Execution continues ...");
  }
}
```

Error Handling Options

- **Option 1:** Fixing using an if statement (QuotientWithIf.java)
- **Option 2:** Fixing using a method (QuotientWithIf.java)
- **Option 3:** Fixing with Exception (QuotientWithException.java)
  - It enables a method to throw an exception to its caller
  - Without it, a method must handle the exception or terminate the program
- **Option 4:** Additional Advantage (InputMismatchExceptionDemo.java)
  - We could let our program continuously read an input until it is correct

Example Scenario 1: Fixing Using an if Statement

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: "); // Prompt the user to enter two integers
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    if (number2 != 0)
       System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
    else
       System.out.println("Divisor cannot be zero ");
  }
}
```

Example Scenario 1: Fixing Using a Method

```
import java.util.Scanner;
public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);  }
    return number1 / number2;  }  // End of quotient() method

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is " + result);  } // End of main() method
}  // End of class
```

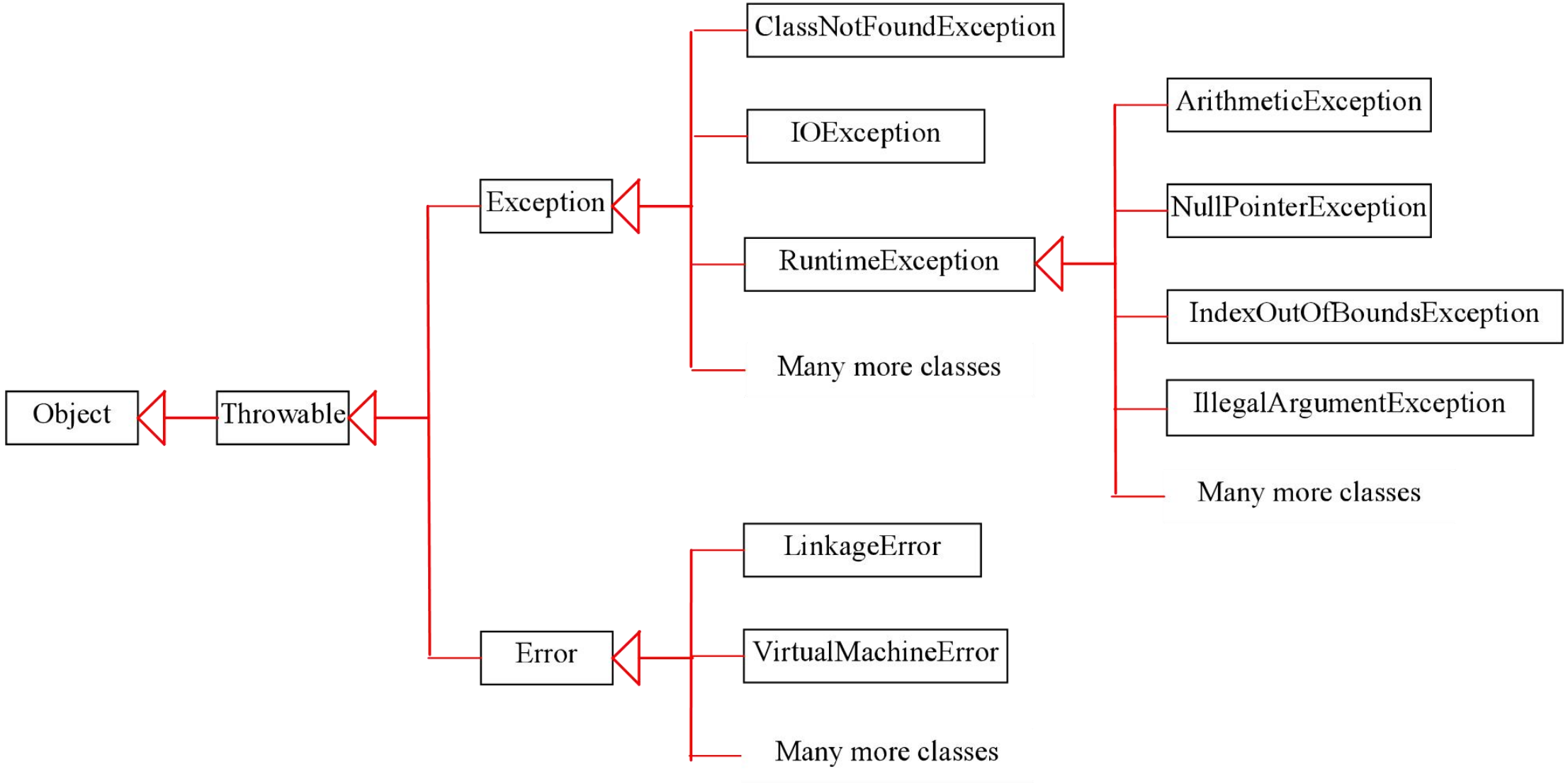
Example Scenario 1: Fixing Using an Exception

```
import java.util.Scanner;
public class QuotientWithException {
    public static int quotient(int number1, int number2) {
        if (number2 == 0) throw new ArithmeticException("Divisor cannot be zero");
        return number1 / number2; }  // End of quotient() method
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();   int number2 = input.nextInt();
        try {
                int result;
                result = quotient(number1, number2);
                System.out.println("This statement and the next will not be executed if exception occurs");
                System.out.println(number1 + " / " + number2 + " is " + result);  } //End of try block
        catch (ArithmeticException ex) {
               System.out.println("Exception: an integer cannot be divided by zero "); } // End of catch
               System.out.println("Execution continues ...");  } // End of main() method
} // End of class
```

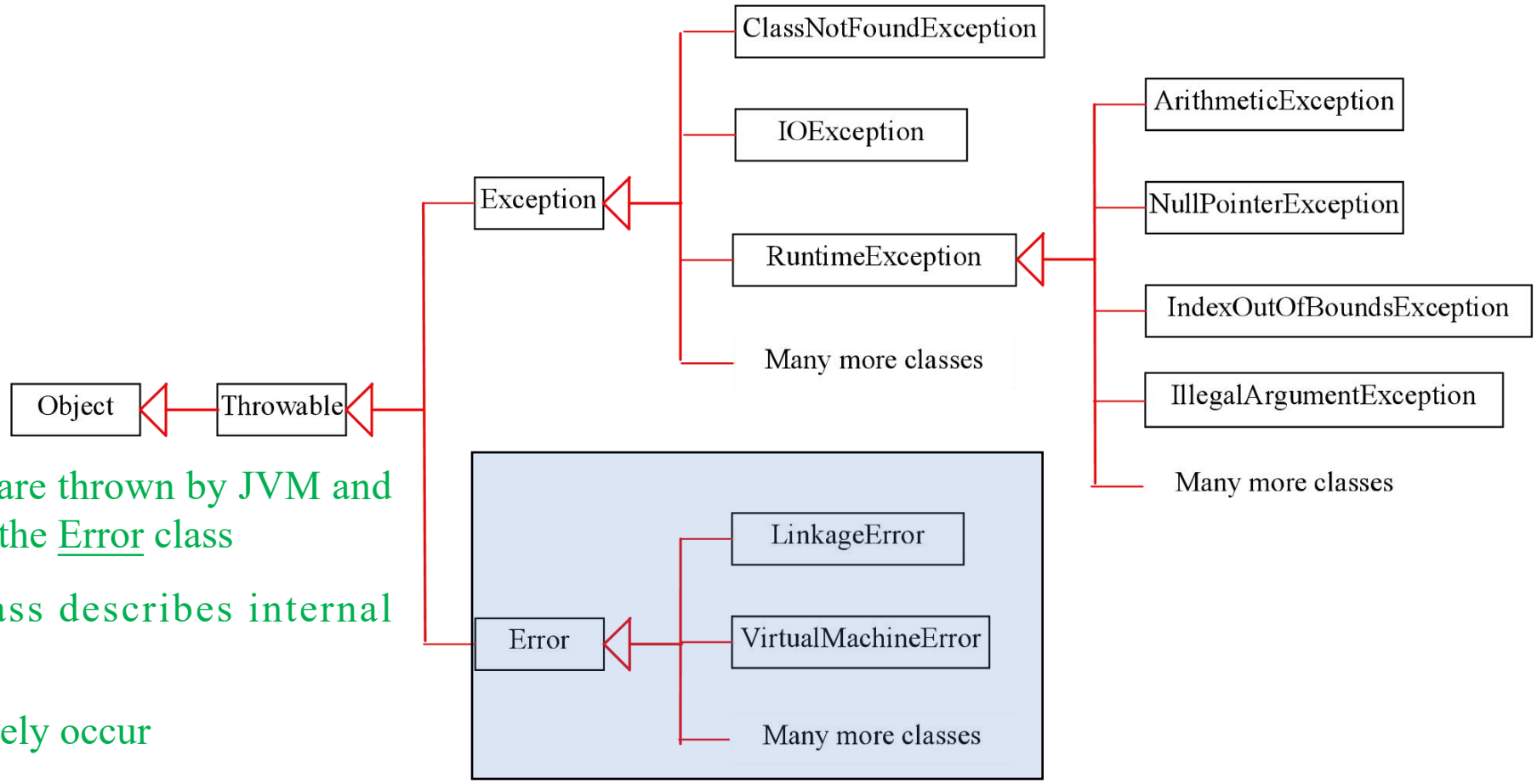
Exceptions in Java

- Many languages, including Java use a mechanism know as *Exceptions* to handle errors at runtime
- In Java, Exception is a **class** with many descendants!
  - ArrayIndexOutOfBoundsException
  - NullPointerException
  - FileNotFoundException
  - ArithmeticException
  - IllegalArgumentException, etc.
- All exception and error types in Java are subclasses of class *Throwable*

Java Exception/Error Hierarchy



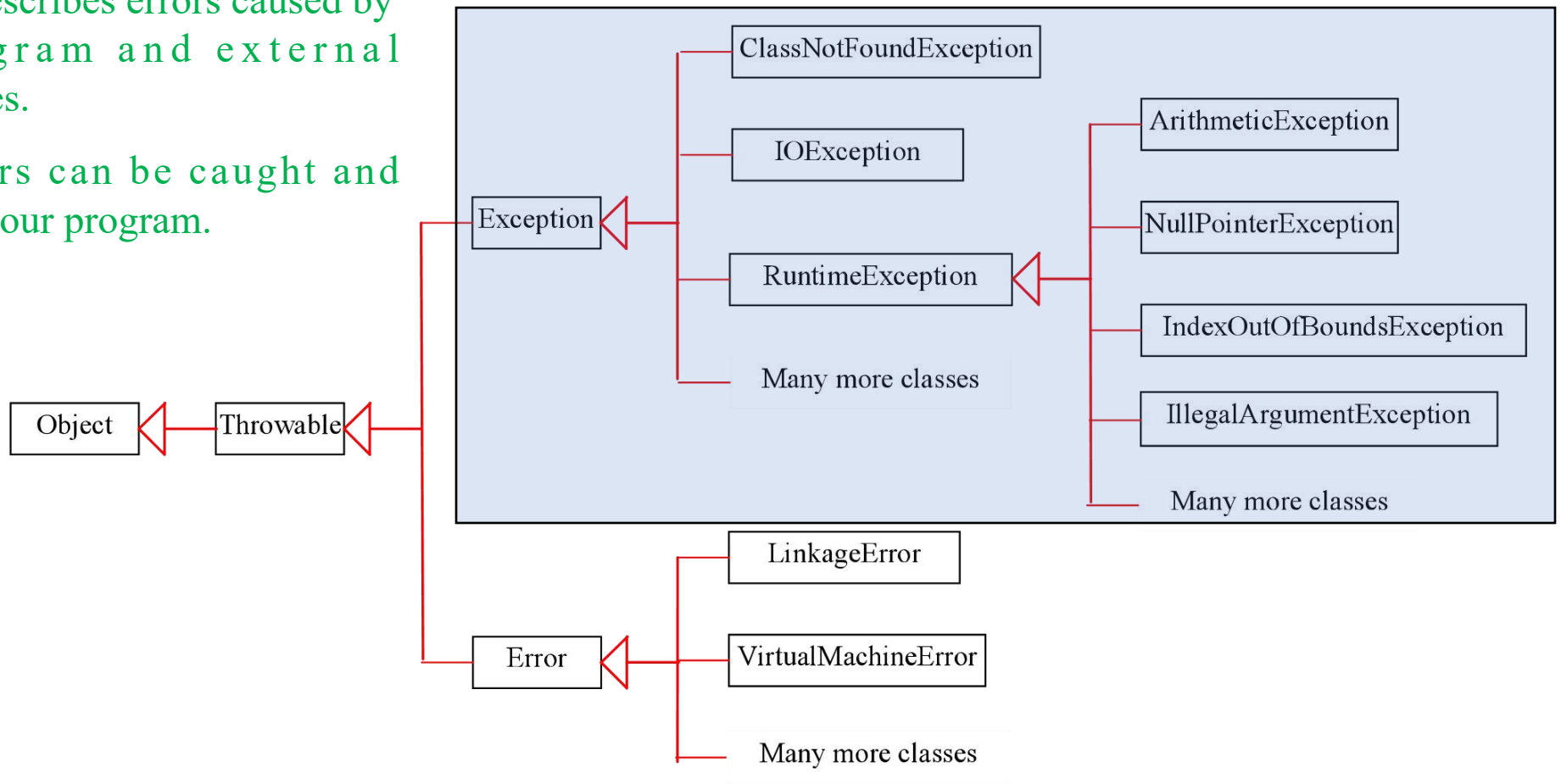
System Errors



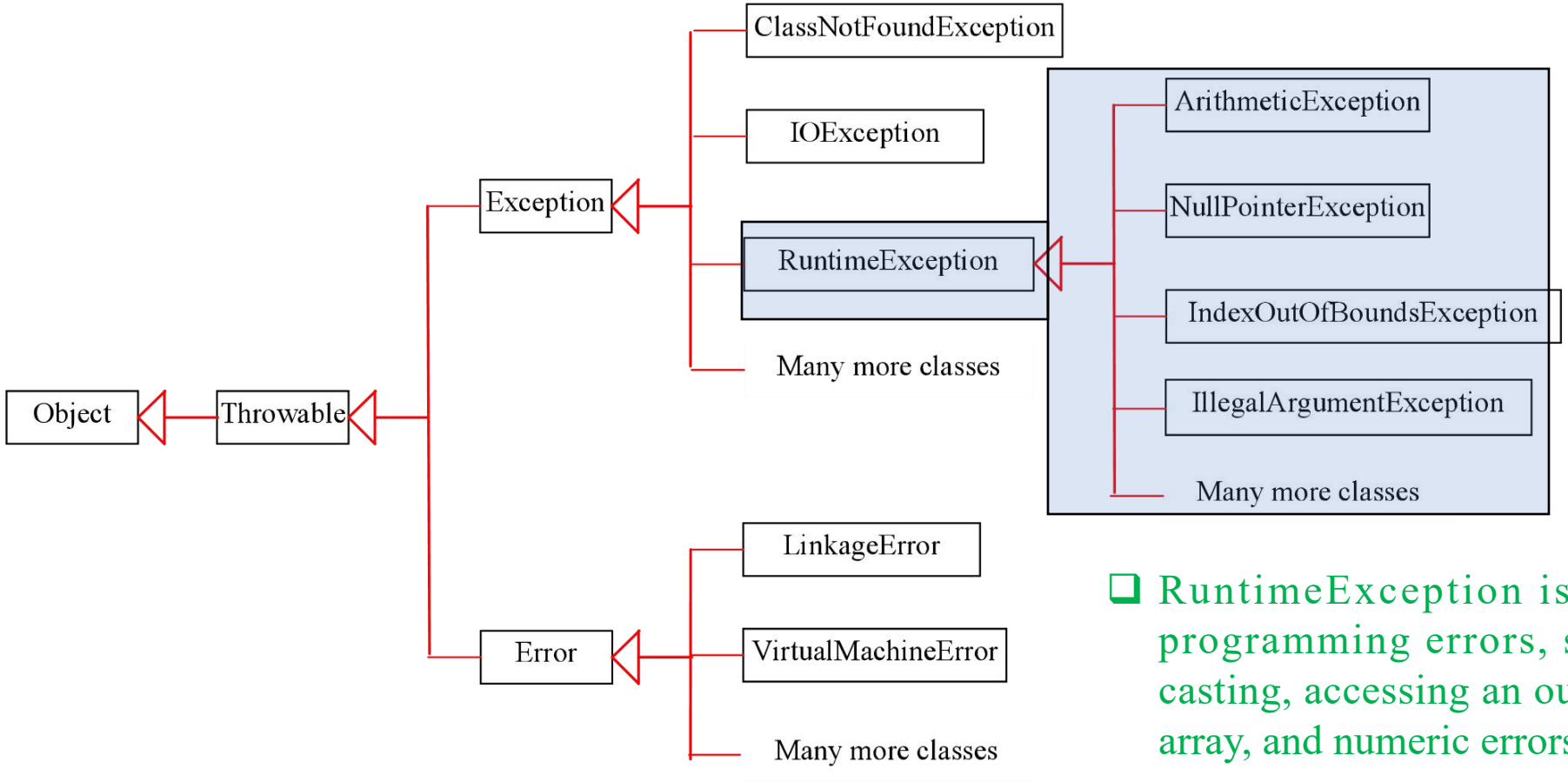
- *System errors* are thrown by JVM and represented in the Error class
- The Error class describes internal system errors
- Such errors rarely occur
- If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully

Exceptions

- Exception describes errors caused by your program and external circumstances.
- These errors can be caught and handled by your program.



Runtime Exceptions



- RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors

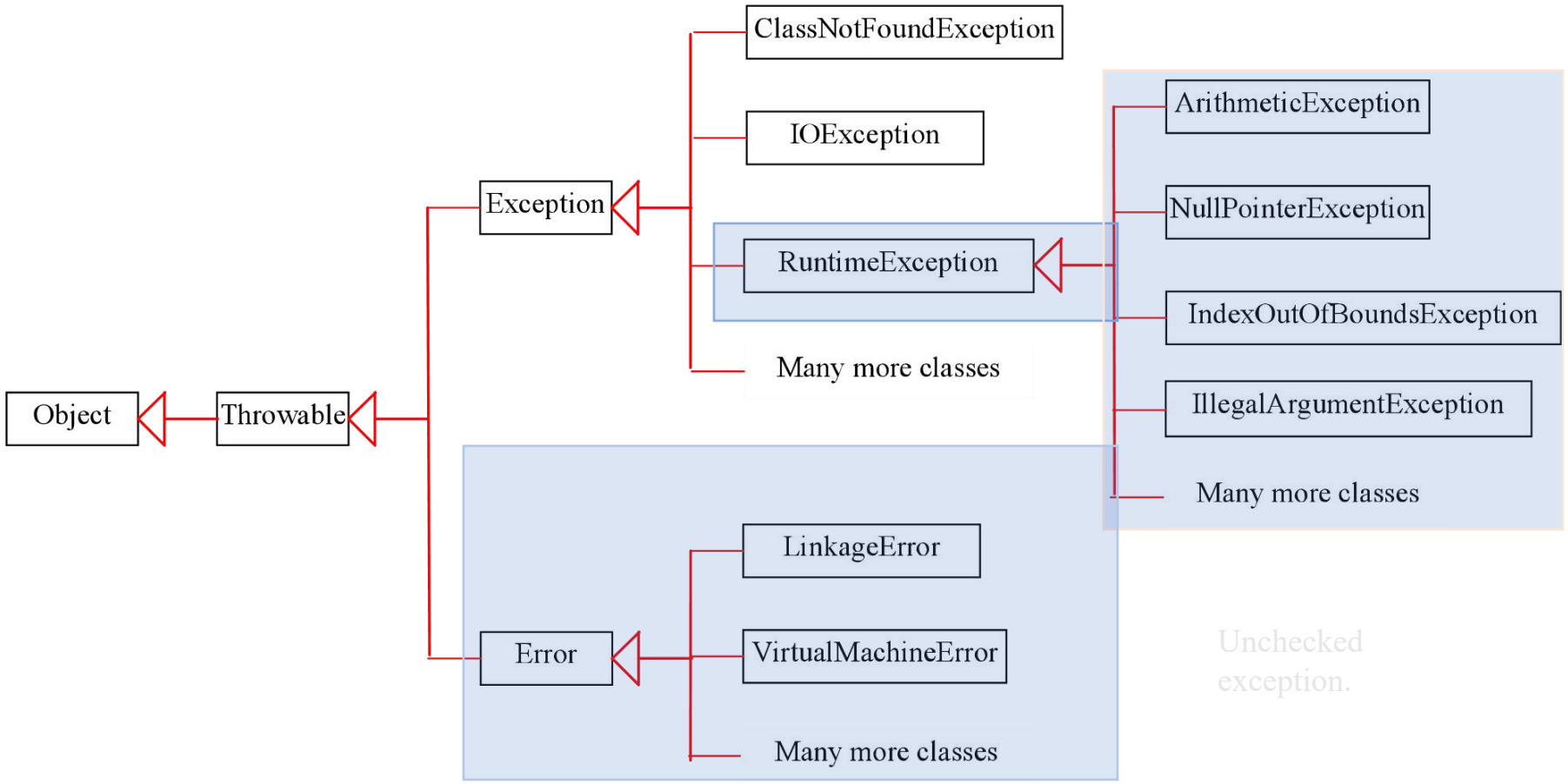
Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*, meaning that the compiler will NOT FORCE the programmer to check and deal with these exceptions
- All other exceptions are known as *checked exceptions*, meaning that the compiler FORCES the programmer to check and deal with these exceptions
  - If not taken care of in the code, the compiler flags ***compilation error*** for ***checked exceptions***!!!

Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example,
  - A NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it
  - An IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array
- These are the logic errors that should be corrected in the program
- Unchecked exceptions can occur anywhere in the program
- To avoid cumbersome overuse of *try-catch* blocks, Java does not mandate us to write code to catch unchecked exceptions

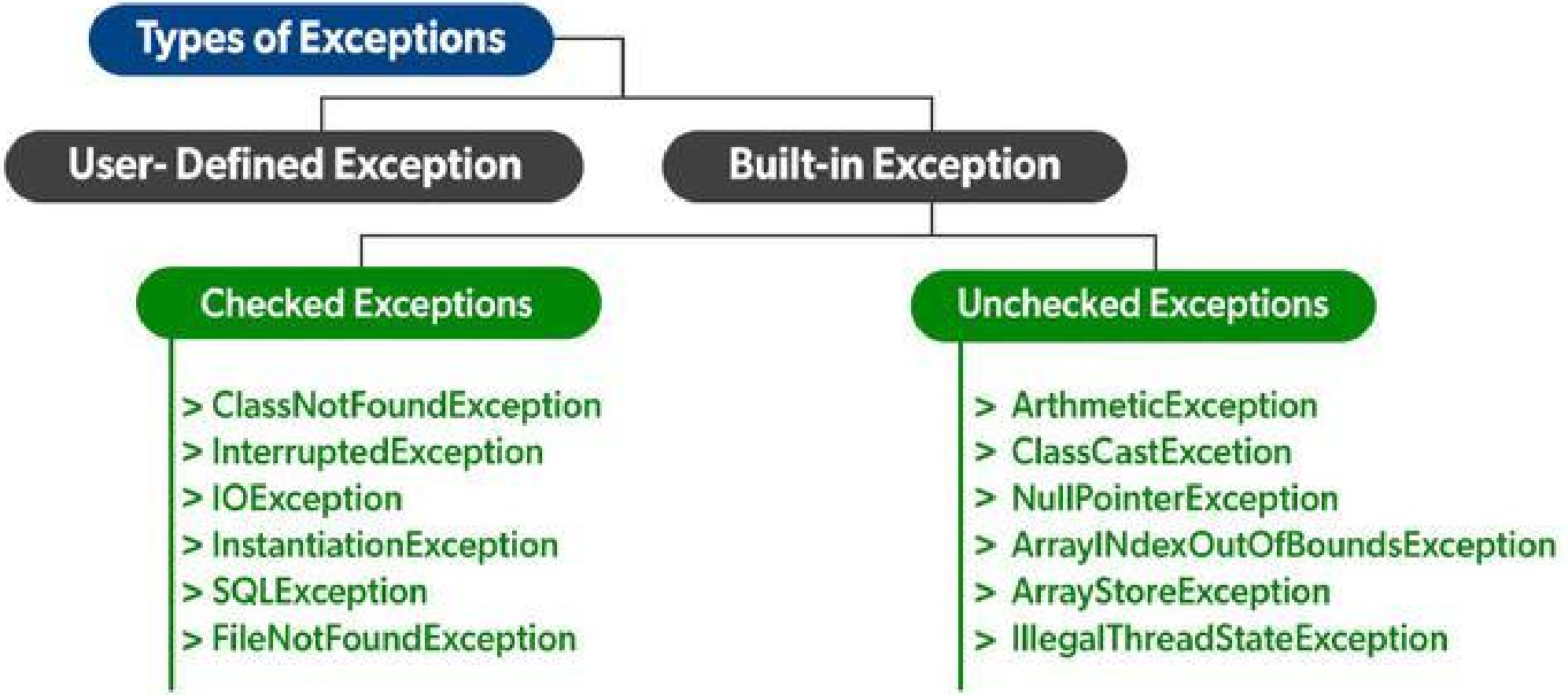
Unchecked Exceptions



Handling Checked Exceptions

- Let us recall that *an **exception** is an event that disrupts the normal flow of the program*
  - *In Java, it is an **object** that is thrown at runtime*
- **Checked exceptions**, unlike Unchecked exceptions (that occurs due to a programming logic error), represent errors that are unpreventable by the programmers!
- Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur, and if they do occur must be dealt with in some way [other than the program terminating]
- Checked exceptions are **checked** at compile time, meaning that, not handling exceptions of this types would incur compilation error

Exception Categories



Exception: Displaying a Description

- Throwable overrides the toString() method defined by Object Class
  - That overridden method returns a string containing a description of the corresponding exception
  - A println() statement can print that description if we simply pass the exception as an argument
  - Example:

```
catch (ArithmeticzException ex) {
        System.out.println("Exception: " + ex);
}
```

Handling Exception in Java

Java Exception handling is managed via five keywords:

- **try:** allows us to define a block of code to be tested for errors while it is being executed
- **catch:** allows us to define a block of code to be executed, if an error occurs in the try block
- **throw:** allows us to explicitly throw a single exception; this statement is used together with an exception type
- **throws:** used to declare the type of exceptions that might occur within the method; used in the declaration of a method
- **finally:** lets us execute a block of code, after try...catch, regardless of the result

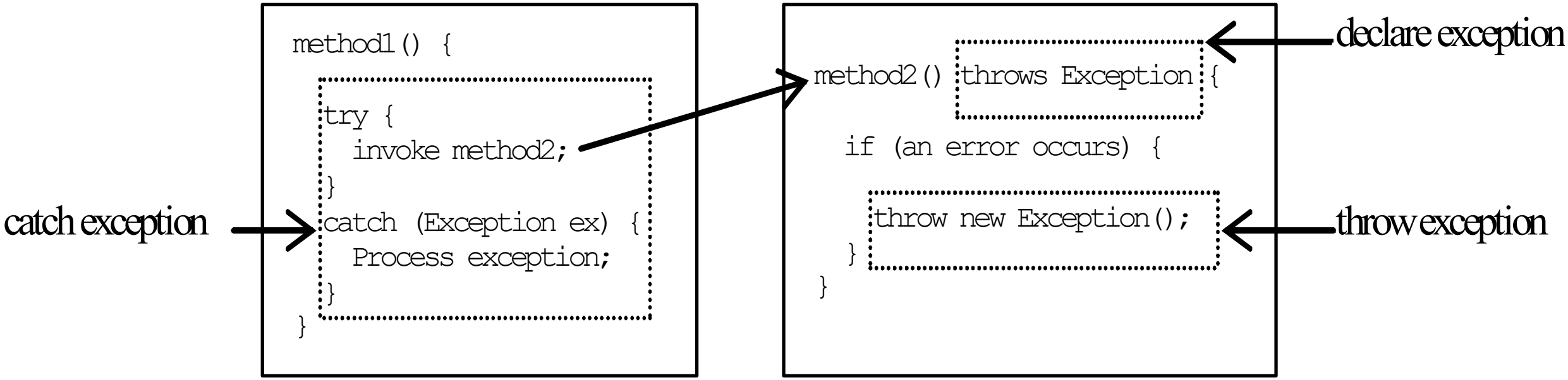
Programmer's Life: Without Exception Handling

```java
import java.util.*;

public class InputMismatchWithoutExceptionDemo {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        //Ask for an integer input
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println("The number entered is " + number);
    }
}
```

Programmer's Life: With Exception Handling

```
import java.util.*;
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;
    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("The number entered is " + number);
        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" + "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```

Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

- Every method must state the types of checked exceptions it might throw
  - known as *declaring exceptions*

- If a method does not handle a checked exception, the method must declare them using the throws keyword
- The throws keyword appears at the end of a method's signature

```
public void myMethod() throws IOException
```

- A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas

```
public void myMethod() throws IOException, OtherException
```

- Essentially, the ***throws keyword*** is used to postpone the handling of a checked exception!

Throwing Exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it
  - known as *throwing an exception*.
- Unlike throws keyword, throw is used to invoke an exception explicitly
- During the execution of a method, if a situation occurs that is to be handled by exceptions, then an Exception is thrown as follows:
  - An Exception object of the proper type is created
  - Flow of control is transferred from the current block of code to code that can handle or deal with the exception
  - Normal flow of the program stops and error handling code takes over (if it exists)
- Example:
  - throw new TheException();

  OR
  - TheException ex = new TheException();

    throw ex;

Throwing Exceptions: Example

```
public class CircleWithException {
        private double radius;
        public CircleWithException(double radius) {
                setRadius(radius);
        }
        public void setRadius (double radius) throws IllegalArgumentException {
                if (radius >= 0)
                        this.radius = radius;
                else
                        throw new IllegalArgumentException("Radius cannot be negative");
        }
}
```

Using `try-catch` Blocks

- If you want to handle a checked exception locally, then use the keywords `try` and `catch`
  - The code that could cause an exception is placed in a block of code preceded by the keyword `try`
  - The code that will handle the exception, if it occurs, is placed in a block of code preceded by the keyword `catch`

```
try {
        statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
        handler for exception1;
}
catch (Exception2 exVar2) {
        handler for exception2;
}
...
catch (ExceptionN exVarN) {
        handler for exceptionN;
}
```

Print The Exception Information

- printStackTrace(): This method prints exception information in the format of -

  “Name of the exception: description of the exception, stacktrace.”

- toString(): This method prints exception information in the format of - “Name of the exception: description of the exception”

- getMessage(): This method prints only the ”description of the exception”

Remember: catch statement is not a “method call”!

```
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result;
      result = quotient(number1, number2);
      System.out.println("This statement and the next will not be executed if exception occurs");
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

Multiple catch Clauses

- More than one exception could be raised by a single piece of code!
- We can specify two or more catch clauses, each catching a different type of exception
- When an exception is thrown
  - each catch statement is inspected in order
  - first one whose type matches that of the exception is executed
  - After one catch statement executes, the others are bypassed
  - execution continues after the try / catch block
- **Note:** Subclass exception types must precede the superclass exception types
  - **Reason:** a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses
  - **Results in Compilation Error:** Unreachable code!

Multiple catch Clauses: Example

```
class MultipleExceptionCatches {
    public static void main(String[] args) {
        try{
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = b;
        }
        catch (ArithmeticException ex) {
            System.out.println("Exception 1: " + ex);
        }
        catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Exception 2: " + ex);
        }
        System.out.println("Execution After try...catch block");
    }
}
```

Mechanics of try and catch

```
public int countChars(String fileName){
	int total = 0; /* counts the characters read */
	try{
		FileReader r = new FileReader(fileName);
		while( r.ready() ){
			r.read();
			total++;
		}
		r.close();
	}
	catch(FileNotFoundException e){
		System.out.println("File named "
			+ fileName + " not found. " + e);
		total = -1;
	}
	catch(IOException e){
		System.out.println("IOException occured " +
			"while counting chars. " + e);
		total = -1;
	}
	return total;
}
```

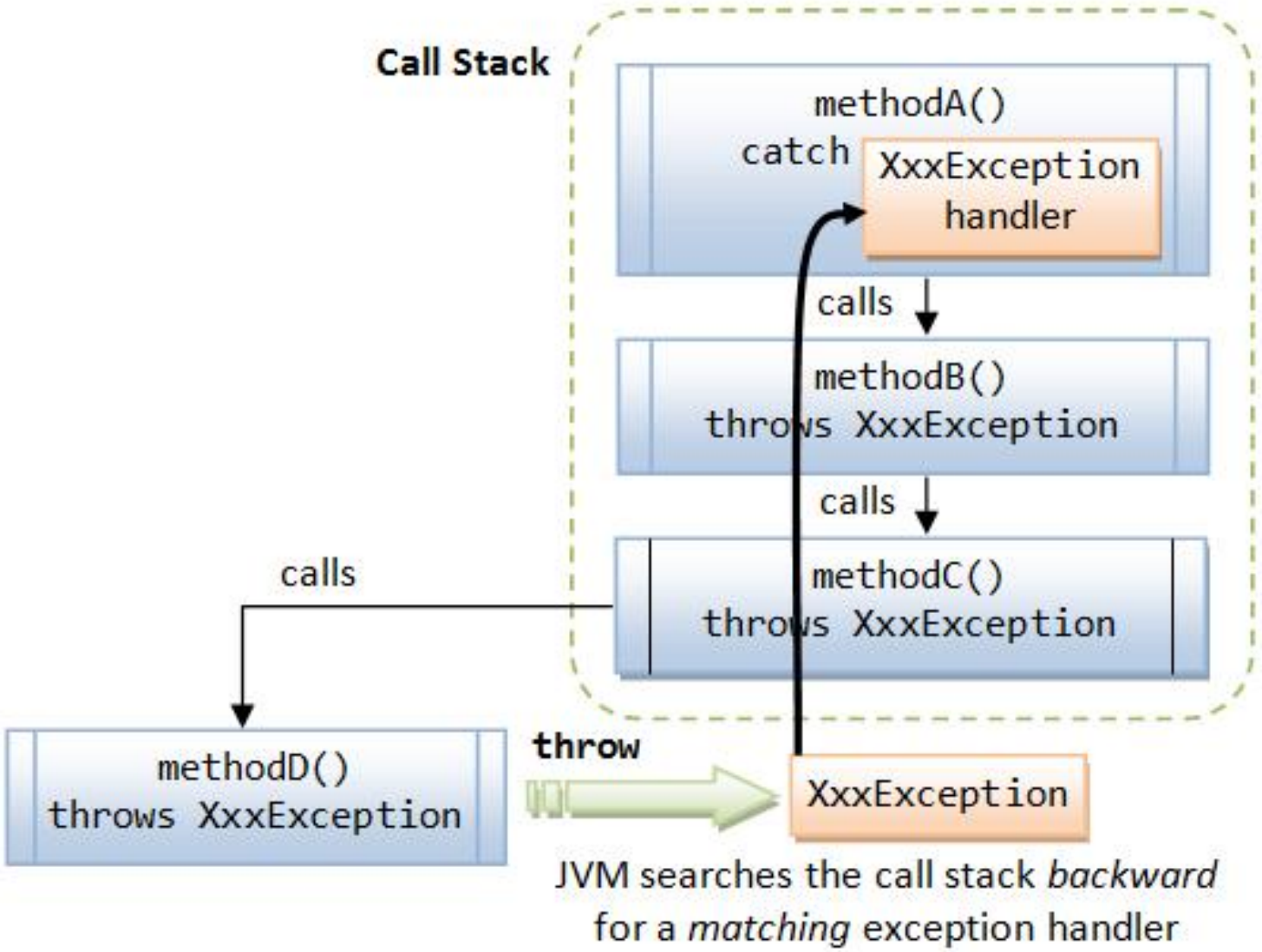
Mechanics of `try` and `catch`

- In the code on the previous slide there are, in fact, four statements that can generate checked exceptions:
  - The FileReader() constructor
  - The ready() method
  - The read() method
  - The close() method
- To deal with the exceptions, we either can state that this method throws an Exception of the proper type or can handle the exception within the method itself
- If we choose the later option, we place the code that could cause the checked exception in a try block
  - Note how the statements are included in one try block
  - Each statement could be in a separate try block with an associated catch block, but that is very inconvenient!
- Each try block must have one or more associated catch blocks
  - Code here to handle the corresponding error
  - In this case, we just print out the error and set result to -1
  - However, more complicated error handling approach may be taken

Mechanics of Java Exception Handling

- If an exception has occurred inside a method, the method creates an Exception Object and hands it off to the run-time system (JVM)
  - The exception object contains the name and description of the exception, and the current state of the program where the exception has occurred
  - Run-time system searches the call stack to find the method that contains the ***Exception Handler***, a block of code that can handle the occurred exception
  - The search proceeds through the call stack in the reverse order in which methods were called
  - If JVM finds an appropriate handler, it passes the occurred exception to that handler
  - If JVM finds no appropriate handler, it pass on the Exception Object to the Default Exception Handler
- The default exception handler is part of the run-time system
- This default exception handler simply prints the exception information and terminates the program abnormally
- When the catch block code is completed, the program DOES NOT "go back" to the code where the exception occurred
  - It finds the next regular statement after the catch block

Mechanics of Java Exception Handling



Catch or Declare Checked Exceptions

Suppose a method p2() is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

- Java forces us to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException),
  - Either we invoke it in a try-catch block, OR
  - We declare to throw the exception in the calling method

Catch or Declare Checked Exceptions: Example

- Suppose that method p1() invokes method p2() mentioned earlier
- Then, code for method p1() must be EITHER as shown in (a) OR as shown in (b)

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

`finally` Clause

- It is possible for an exception to cause the method to return prematurely
  - May cause Resource Leak
  - Caused when resources are not released by a program
  - Files, Database Connections, Network Connections, etc.
- The `finally` keyword (optional) is designed to address the above contingency
  - The finally block appears after the catch blocks
  - Always executes whether or not any exception is thrown
  - Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns
  - So, may be used to release resources (e.g., close file), if any

The `finally` Clause

```
try {
   statements;
}
catch(TheException ex) {
   handle ex;
}
finally {
   finalStatements;
}
```

Rethrowing Exceptions

```
try {
   statements;
}
catch(TheException ex) {
   perform operations before exit;
   throw ex;
}
```

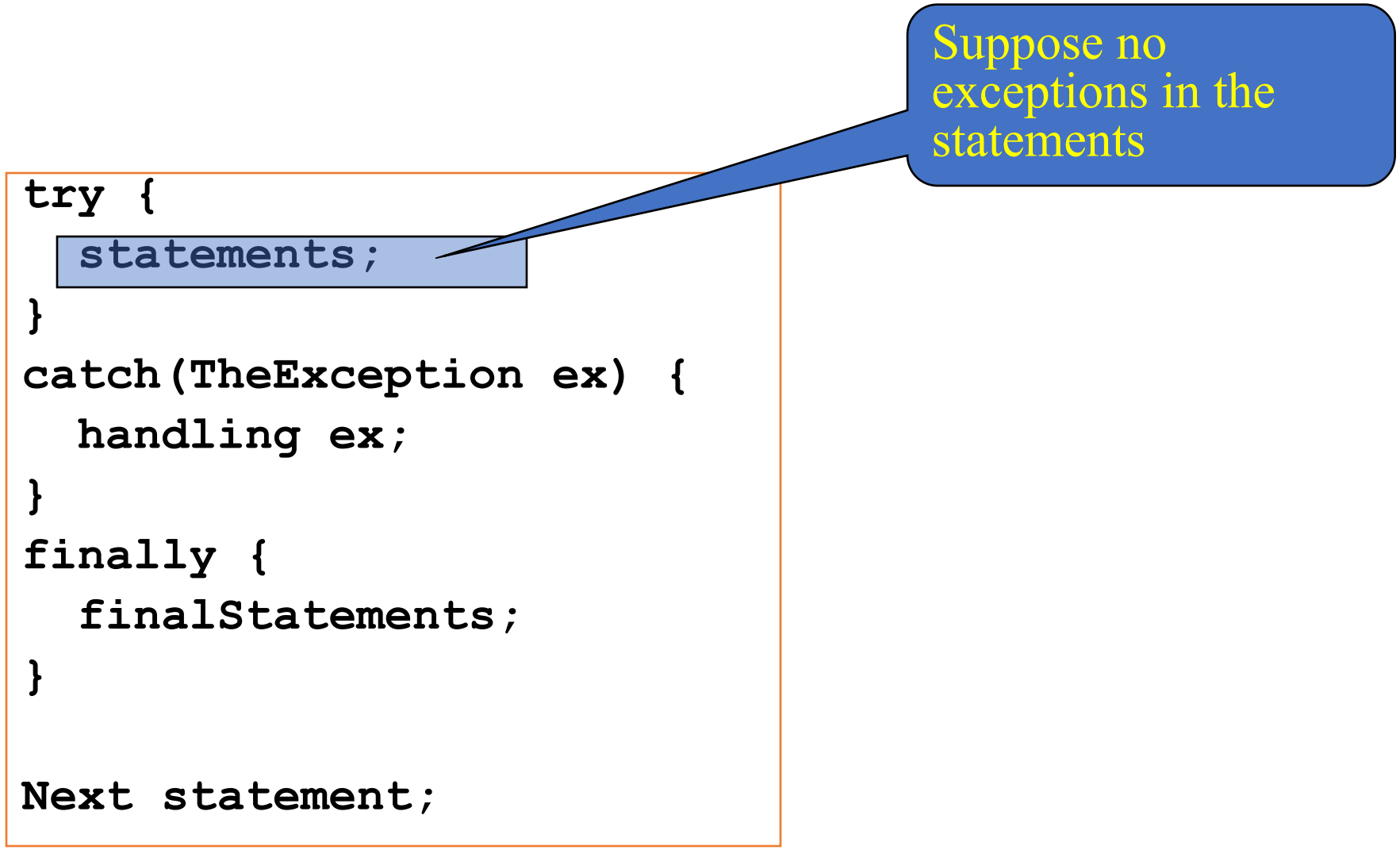
Rethrowing Exceptions: Example

```
class RethrowDemo {
    static void demoproc () {
        try {
            throw new NullPointerException("Demo");
            //System.out.println("This is an unreachable statement.");
        }
        catch (NullPointerException ex) {
            System.out.println("Caught inside demoproc() method." + ex);
            throw ex;
        }
    }

    public static void main (String[] args) {
        try {
            demoproc();
        }
        catch (NullPointerException ex) {
            System.out.println("Recaught inside main() method." + ex);
        }
    }
}
```

Methods in *Throwable* Class

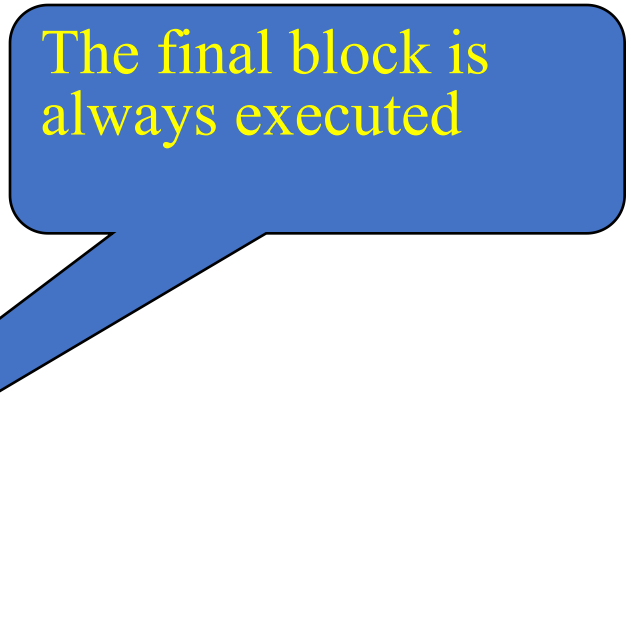
- Methods in Throwable class
  - Method printStackTrace()
    - Prints method call stack (helpful in debugging)
    - Prints the throwable along with other details like the line number and class name where the exception occurred
  - Method getStackTrace()
    - Obtains stack-trace information in an array of stack trace elements
    - Array elements are same as the stack trace information printed by printStackTrace()
    - Each element in the array represents one stack frame
  - Method getMessage()
    - Returns descriptive string of the exception
    - The same may then be printed using print() or println() statement
    - Returns null if no such string is available for the exception
  - Method toString()
    - Returns a String object containing a description of the exception
    - Called by println() when we output a throwable object
- **Note:** Default Exception Handler: Displays complete stack trace

Trace a Program Execution



Trace a Program Execution

```
try {
   statements;
}
catch(TheException ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

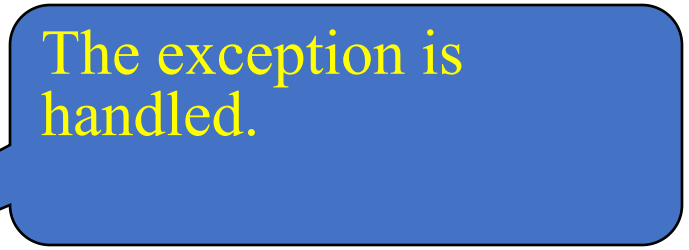
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

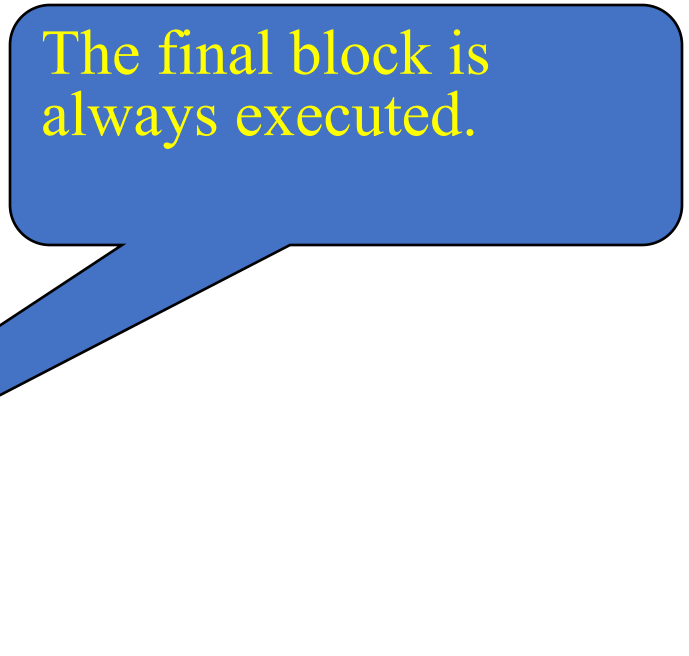
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

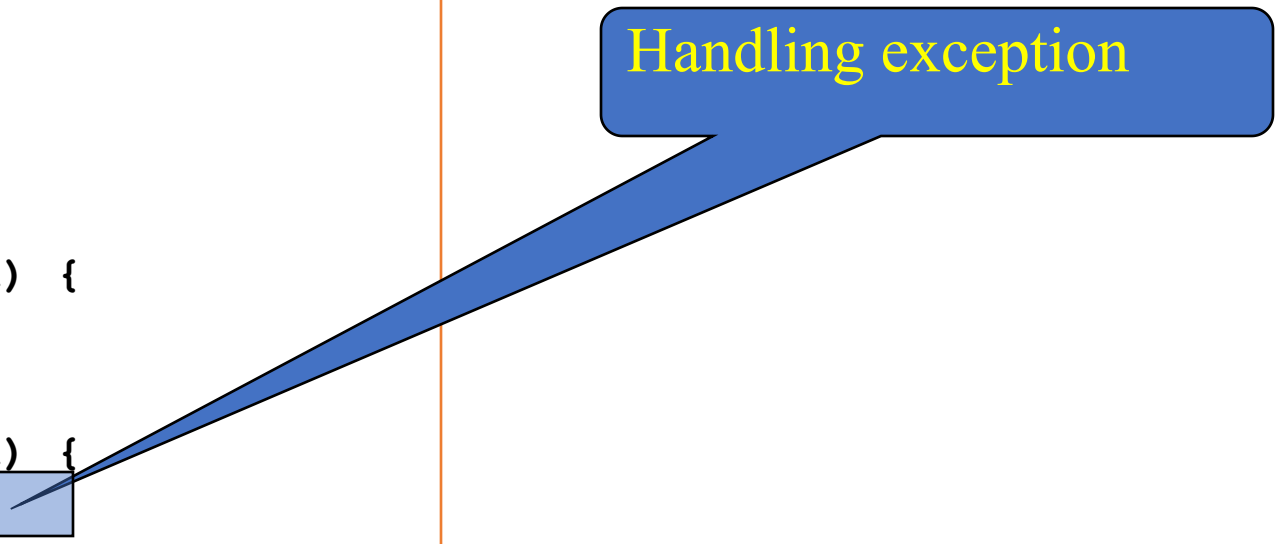
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

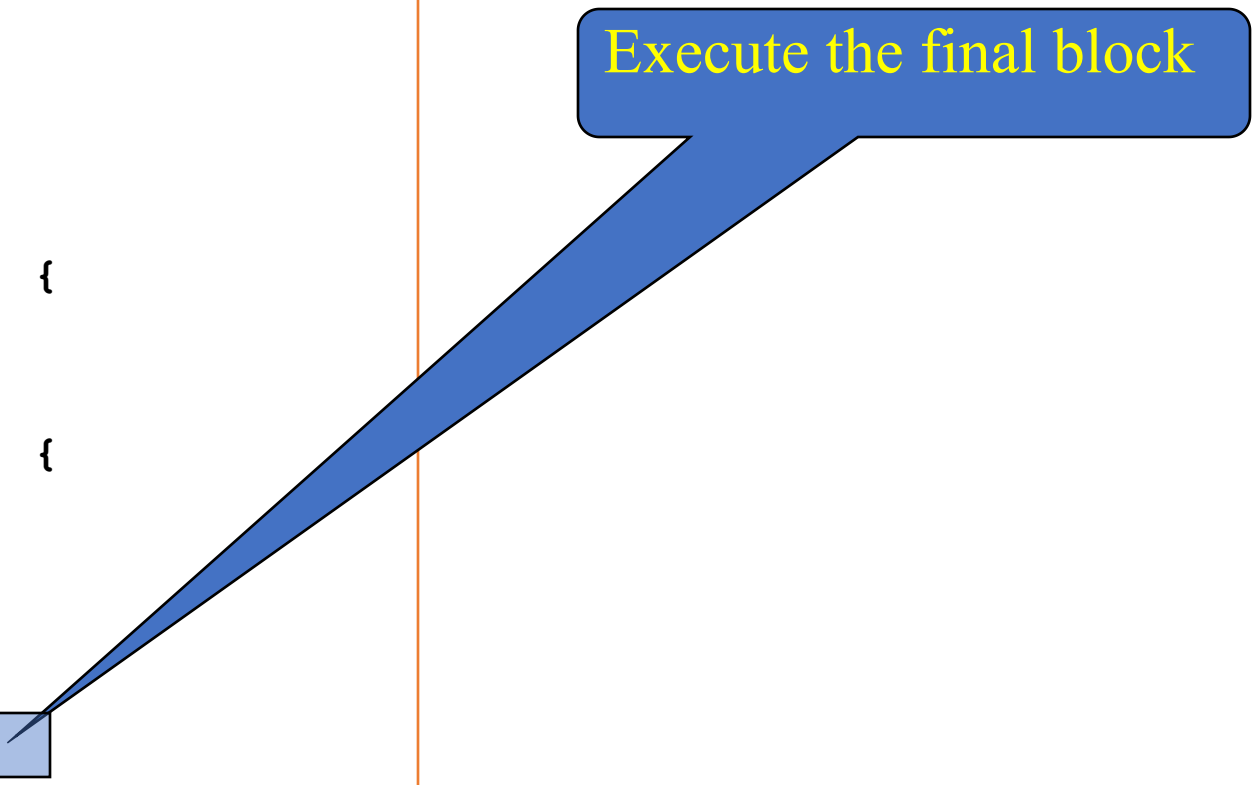
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

Java Built-in Exceptions

- Inside the standard package java.lang, Java defines several exception classes:

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**

Defining Custom Exception Classes

- Use the exception classes in the program whenever possible
- Define custom exception classes if the predefined classes are not sufficient
- Define custom exception classes by extending Exception or a subclass of Exception
- The Exception class does not define any methods of its own
  - Indeed, it inherit methods provided by Throwable
- Specifying a description when an exception is created is often useful
  - Either can pass as an argument to the Exception constructor
  - OR can override the toString() method associated with Exception class

Custom Exception Class: Example

```
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

  /** Return the radius */
  public double getRadius() {
    return radius;
  }
}
```

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks
  - Makes programs easier to read and to modify
- However, exception handling usually requires more time and resources!
  - It requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods
  - So, exception handling should be made with caution
  - Java mandates exception handling only for checked exceptions
  - For unchecked exceptions, more straight-forward approaches are preferred

When to Throw Exceptions

- An exception occurs in a method
- If we want the exception to be processed by its caller, you should create an exception object and throw it
- If you can handle the exception in the method where it occurs, there is no need to throw it

When to Use Exceptions

- When should we use the try-catch block in the code?
  - We should use it to deal with unexpected error conditions.
  - We must not use it to deal with simple, expected situations. For example, consider the following code:

```
try {
   System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
   System.out.println("refVar is null");
}
```

When to Use Exceptions

- The above code should better be replaced by the following:

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Constructors and Exception Handling

- Constructors cannot return a value to indicate an error
- Throw exception if constructor causes error
  - For example, if invalid initialization value given to constructor and there is no sensible way to correct this

List of Common Checked Exceptions in Java

Common checked exceptions defined in the `java.lang` package:

- `ReflectiveOperationException`
  - `ClassNotFoundException`
  - `InstantiationException`
  - `IllegalAccessException`
  - `InvocationTargetException`
  - `NoSuchFieldException`
  - `NoSuchMethodException`
- `CloneNotSupportedException`
- `InterruptedException`

Common checked exceptions defined in the `java.io` package:

- `IOException`
  - `EOFException`
  - `FileNotFoundException`
  - `InterruptedIOException`
  - `UnsupportedEncodingException`
  - `UTFDataFormatException`
  - `ObjectStreamException`
- `InvalidClassException`
- `InvalidObjectException`
- `NotSerializableException`
- `StreamCorruptedException`
- `WriteAbortedException`

Common checked exceptions defined in the `java.net` package (almost are subtypes of `IOException`):

- `SocketException`
  - `BindException`
  - `ConnectException`
- `HttpRetryException`
- `MalformedURLException`
- `ProtocolException`
- `UnknownHostException`
- `UnknownServiceException`

Common checked exceptions defined in the `java.sql` package:

- `SQLException`
  - `BatchUpdateException`
  - `SQLClientInfoException`
  - `SQLNonTransientException`
- `SQLDataException`
- `SQLFeatureNotSupportedException`
- `SQLIntegrityConstraintViolationException`
- `SQLSyntaxErrorException`
  - `SQLTransientException`
- `SQLTimeoutException`
- `SQLTransactionRollbackException`
- `SQLTransientConnectionException`
  - `SQLRecoverableException`
  - `SQLWarning`

List of Common Unchecked Exceptions in Java

Common unchecked exceptions in the `java.lang` package:

- `ArithmeticException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `ArrayStoreException`
- `ClassCastException`
- `EnumConstantNotPresentException`
- `IllegalArgumentException`
  - `IllegalThreadStateException`
  - `NumberFormatException`
- `IllegalMonitorStateException`
- `IllegalStateException`
- `NegativeArraySizeException`
- `NullPointerException`
- `SecurityException`
- `TypeNotPresentException`
- `UnsupportedOperationException`

Common unchecked exceptions in the `java.util` package:

- `ConcurrentModificationException`
- `EmptyStackException`
- `NoSuchElementException`
  - `InputMismatchException`
- `MissingResourceException`

Source Link:
https://www.codejava.net/java-core/exception/java-checked-and-unchecked-exceptions