CENG 1004
Introduction to Object Oriented Programming

Spring 2018

WEEK 13

Exception Handling

How to Handle Errors

- Assume we have a class that converts numbers in word representation to integer .

```
public int convert(String text) ;
```

  - “seventy five” converted to 75

How to Handle Errors

- What should we do if the given input causes errors in methods

```
convert("xyz");
convert("one sixty);
```

Separating Error-Handling Code from "Regular" Code

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Separating Error-Handling Code from "Regular" Code

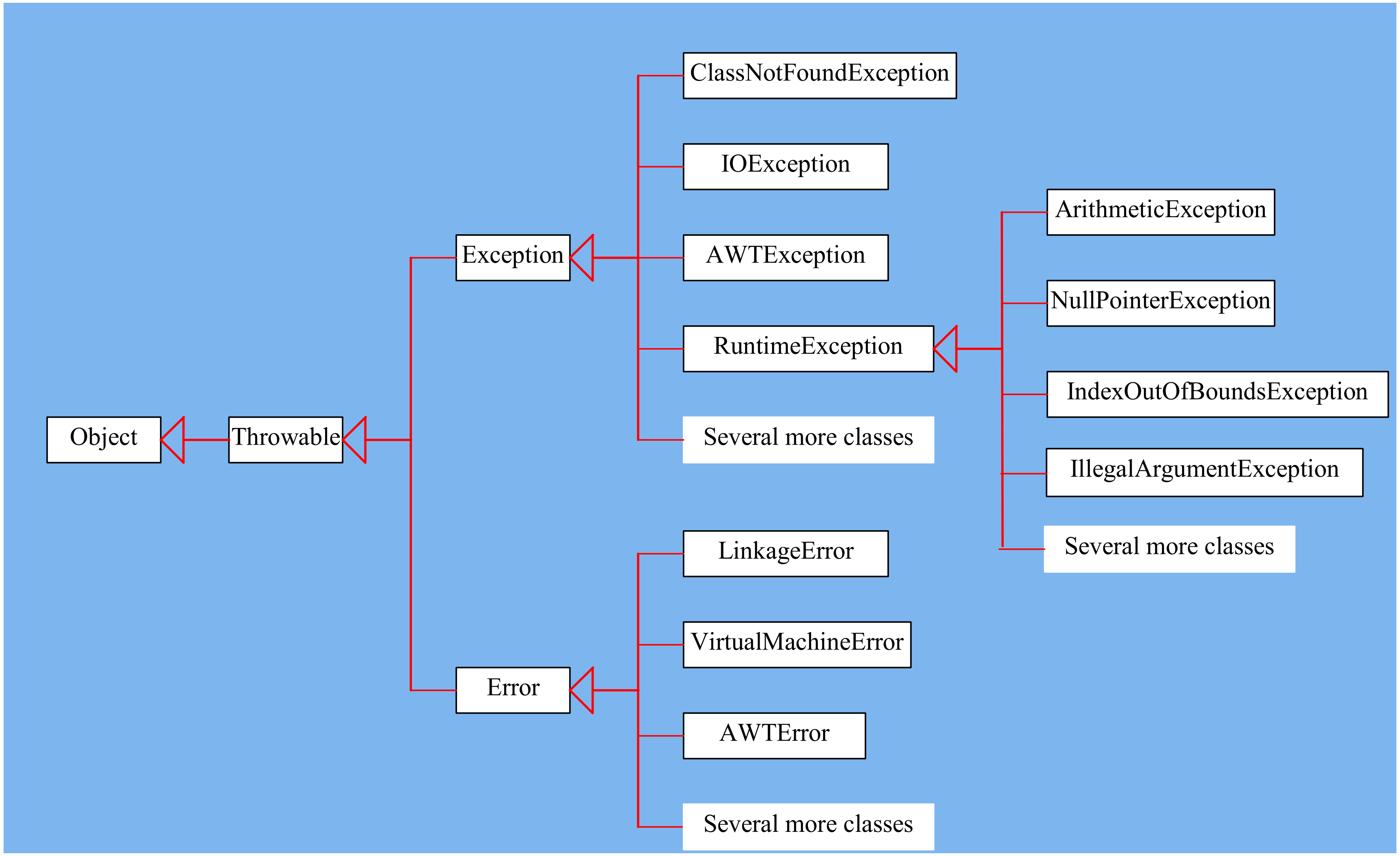
- Exception handling mechanism in Java lets you put all your error-handling code in one easy-to-read place.
- You don't have to use return values to forward errors.
- It also let's you to know that the method you are calling is risky that is the call may results errors
- If you know you might get an exception when you call a method then you can be prepared of the problem.

What Is an Exception?

- The Java programming language uses exceptions to handle errors and other exceptional events.

- **Definition:** An exception is an event, which occurs during the execution of a program, that breaks the normal flow of the program's instructions.

Exception Classes



ConvertException

```
public class ConvertException extends Exception {

   public ConvertException(String number){
         super(number + " can not be converted");
   }
}
```

Throwing an Exception?

- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called **throwing an exception.**

Throwing an Exception

- All methods use the throw statement to throw an exception.
  - The throw statement requires a single argument: a throwable object.
  - Throwable objects are instances of any subclass of the Throwable class.

```
public String convert(String number)  {
    if (notANumber(number)){
          throw new ConvertException(number);
    }
    ...
}
```

Throwing an Exception

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

The Catch or Specify Requirement

- If an exception occurs in a method you should either
  - Catch and handle the exception or
  - Specify the exception to be thrown by the method
- Code that fails to meet the Catch or Specify will not compile
- Not all exceptions are subject to the Catch or Specify Requirement.

The Three Kinds of Exceptions

- Checked exception
  - exceptional conditions that a well-written application should anticipate and recover from.
  - checked for during compile time
  - are subject to the Catch or Specify Requirement
  - All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

Checked exception

- If a checked exception is thrown in a method, it is subject to the Catch or Specify Requirement.

```
public void countWords(String filename){
    BufferedReader br;
    try {
        br = new BufferedReader(new FileReader(filename));
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Checked exception

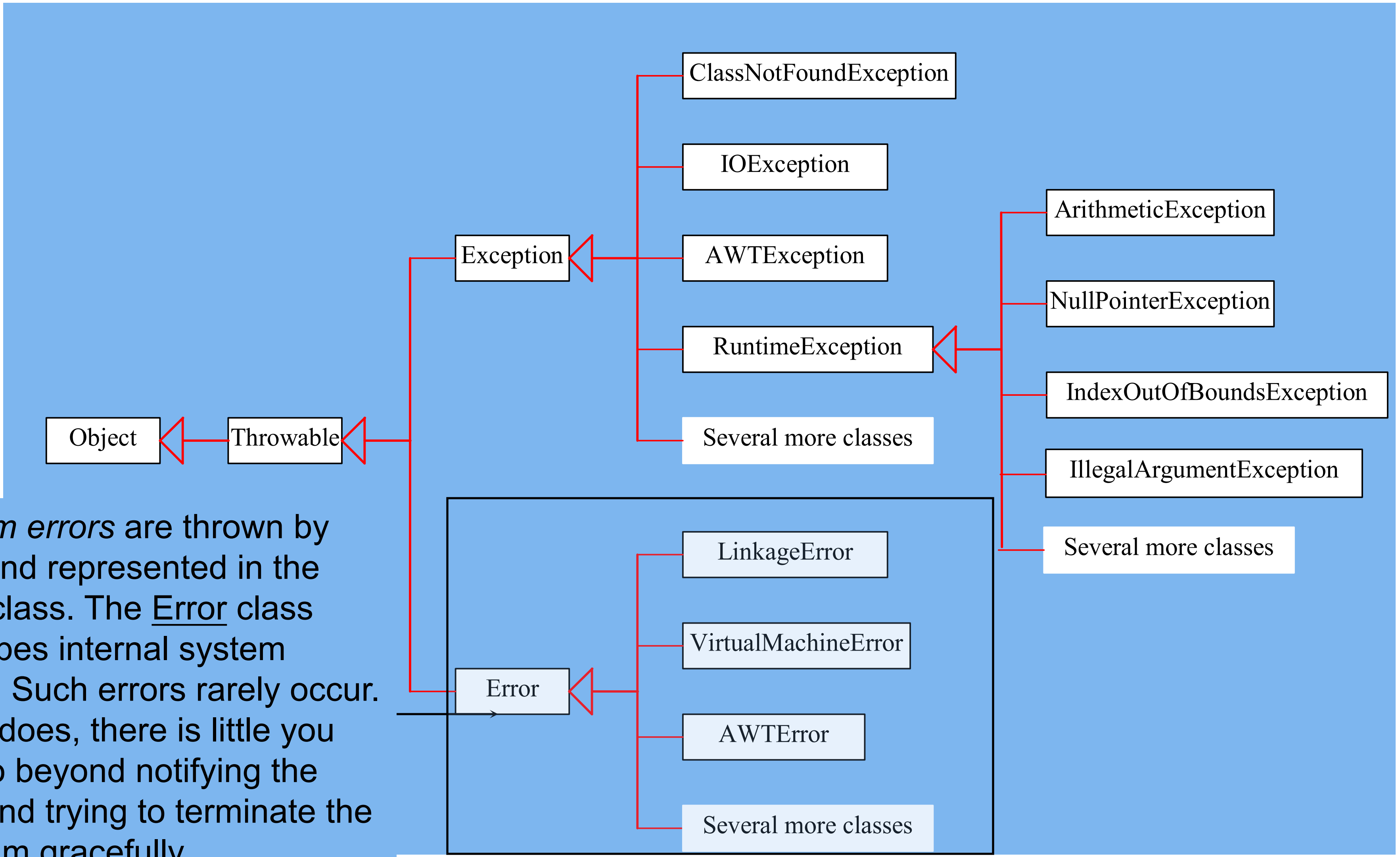
- If a checked exception is thrown in a method, it is subject to the Catch or Specify Requirement.

```
public FileInputStream(File file) throws FileNotFoundException {
    ...
   if (file.isInvalid()) {
      throw new FileNotFoundException("Invalid file path");
   }
  ...
}
```

The Three Kinds of Exceptions

- Error
  - exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
    - An application might choose to catch this exception, in order to notify the user of the problem
    - but it also might make sense for the program to print a stack trace and exit.
  - Errors are not subject to the Catch or Specify Requirement.
    - Errors are those exceptions indicated by Error and its subclasses.

System Errors

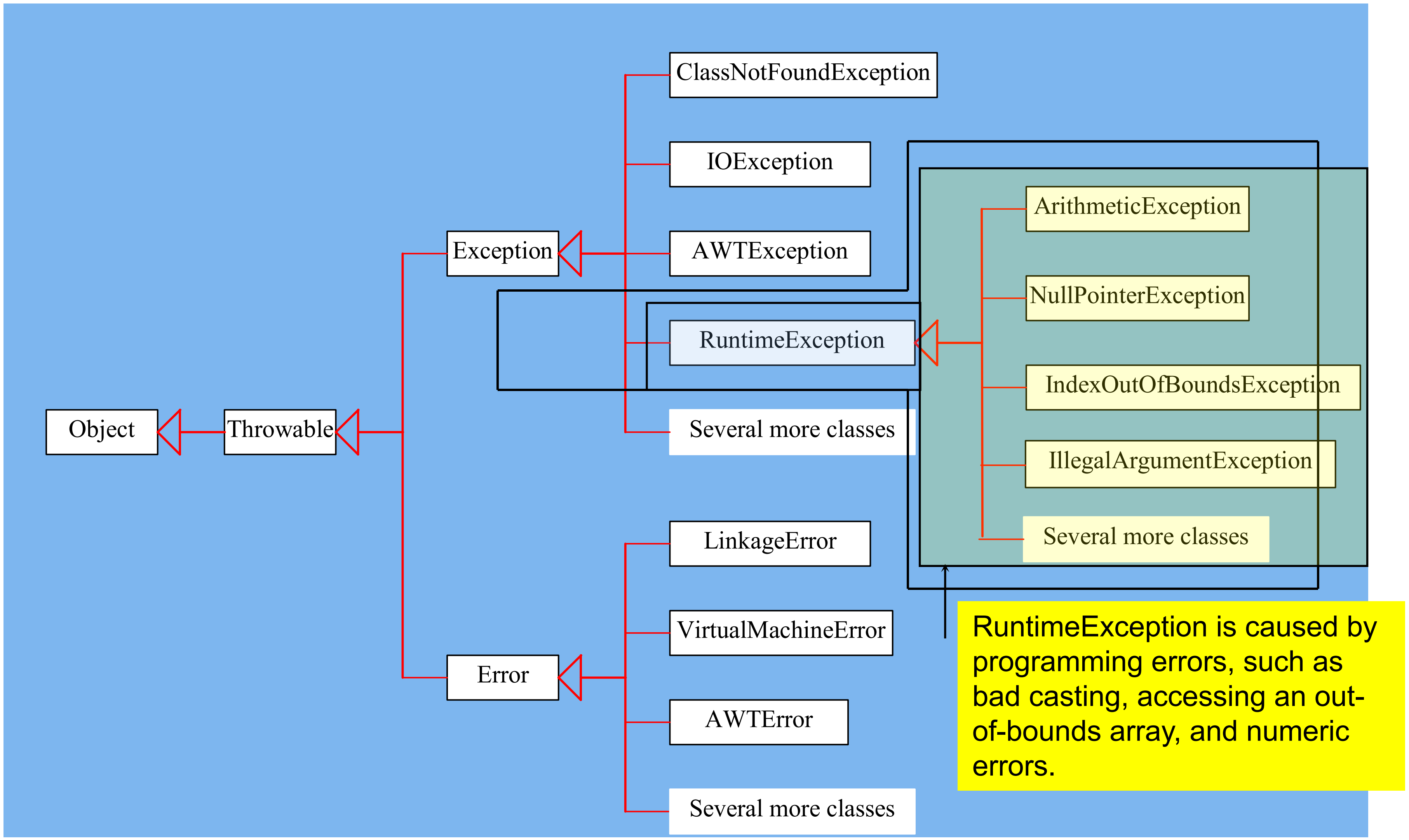


System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

The Three Kinds of Exceptions

- Runtime Exception
  - Exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
  - usually indicate programming bugs, such as logic errors or improper use of an API.
  - Consider the application described previously that passes a file name to the constructor for FileReader.
    - If a logic error causes a null to be passed to the constructor, the constructor will throw NullPointerException.
    - The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.
  - Runtime exceptions are not subject to the Catch or Specify Requirement.
    - Runtime exceptions are those indicated by RuntimeException and its subclasses.

Runtime Exceptions



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

Runtime Exception

```
public FileInputStream(File file) throws FileNotFoundException {
    String name = (file != null ? file.getPath() : null);
    ...
    if (name == null) {
        throw new NullPointerException();
    }
    if (file.isInvalid()) {
        throw new FileNotFoundException("Invalid file path");
    }
    ...
}
```

RuntimeException Class

- reserved for exceptions that indicate incorrect use of an API
  - An example of a runtime exception is **NullPointerException**, which occurs when a method tries to access a member of an object through a null reference.

Checked Exceptions vs. Unchecked Exceptions

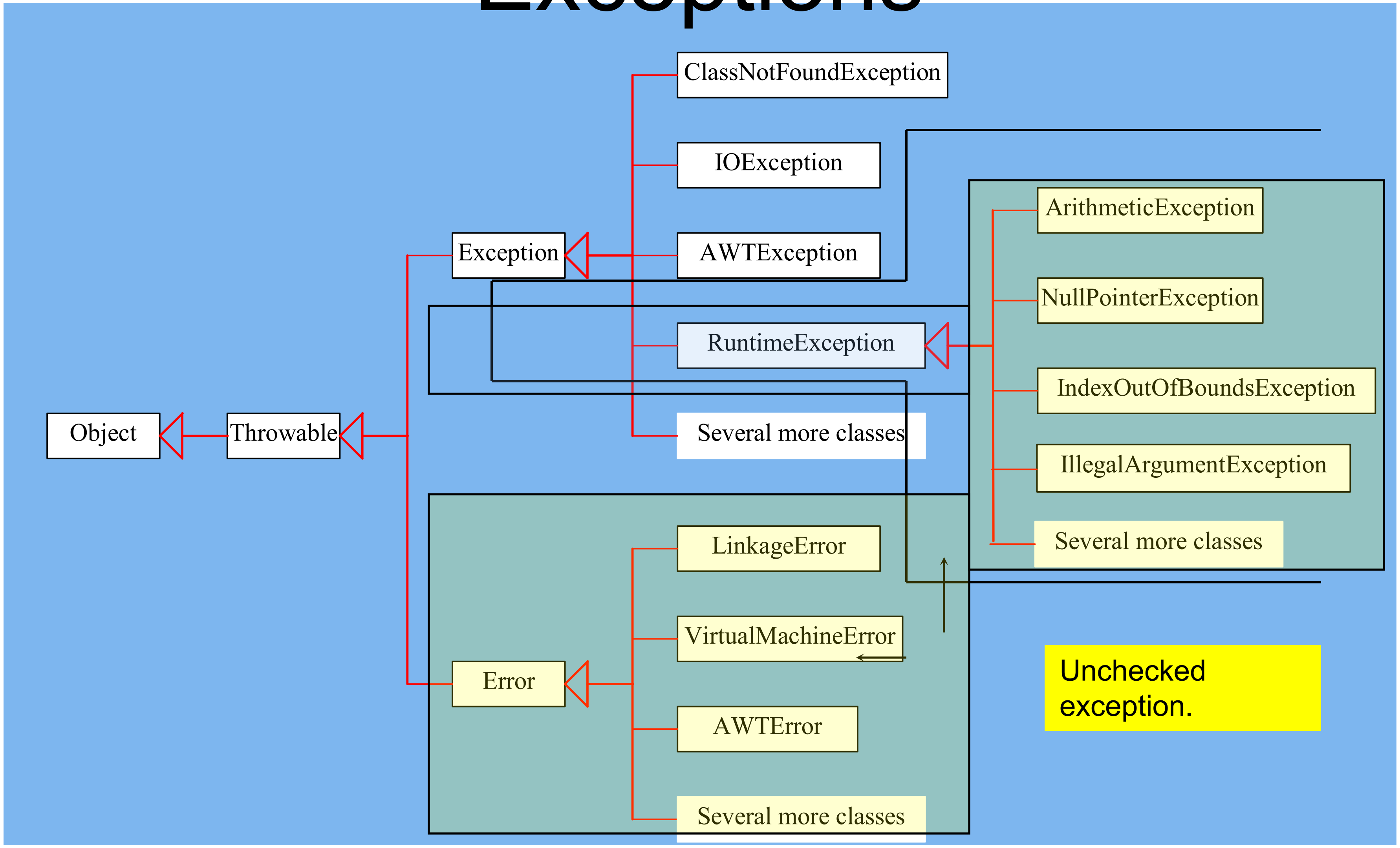
<u>RuntimeException</u>, <u>Error</u> and their subclasses are known as ***unchecked exceptions***.

All other exceptions are known as ***checked exceptions***, meaning that the compiler forces the programmer to check and deal with the exceptions.

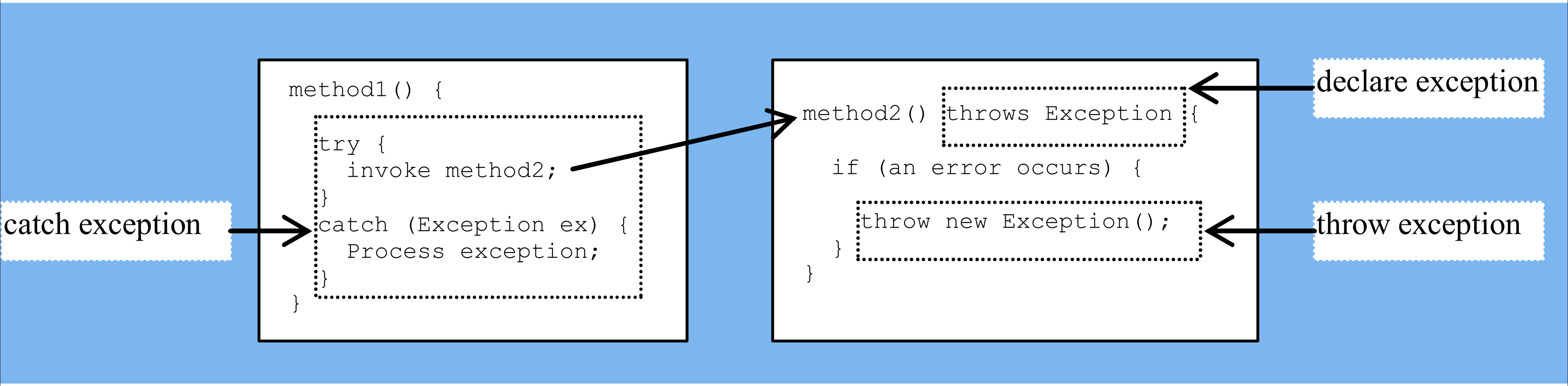
Unchecked Exceptions

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 you to write code to catch unchecked exceptions.**

Checked or Unchecked Exceptions



Declaring, Throwing, and Catching Exceptions



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

Declaring/Specifying Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring/specifying exceptions*.

```
public void myMethod()
   throws IOException

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

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```

Throwing Exceptions Example

```
/** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Handling Exceptions

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.

```
try {
    code
}
catch and finally blocks . . .
```

Handling Exceptions

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

The catch Blocks

- You associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

```
try {

} catch (ExceptionType1 name) {

} catch (ExceptionType2 name) {

}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

The catch Blocks

- Each catch block is an exception handler that handles the type of exception indicated by its argument.
- ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class.
- The handler can refer to the exception with variable name.

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
   File file = new File ("MyFile.txt");
   Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not found.");
}
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.
- Example: [OpenFile.java]

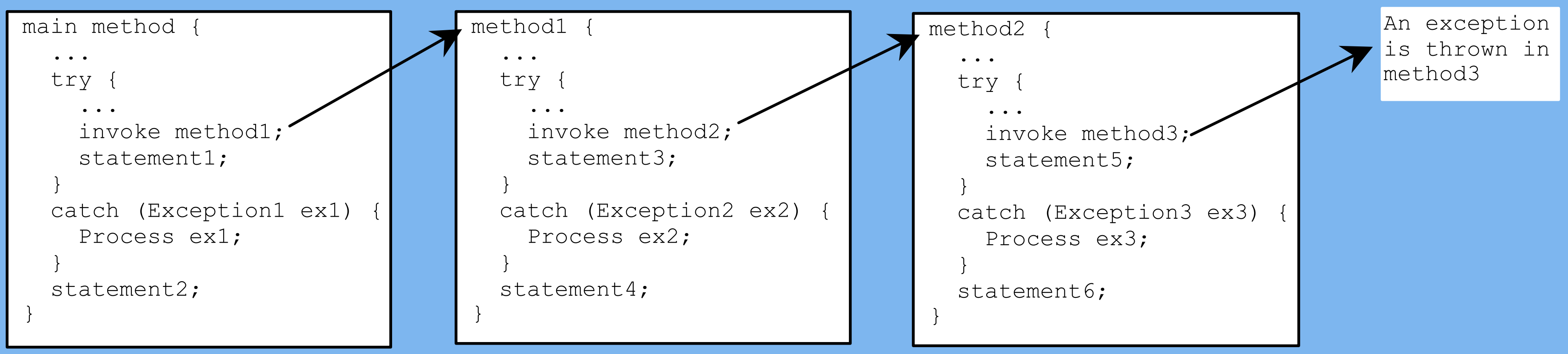
Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception’s type.
- After an exception, the program will continue execution at the point just past the catch block.

The catch Blocks

- Exception handlers can do more than just print error messages or halt the program.
  - They can do error recovery,
  - Prompt the user to make a decision, or
  - Propagate the error up to a higher-level handler using chained exceptions,

Catching Exceptions



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

Catch or Specify Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

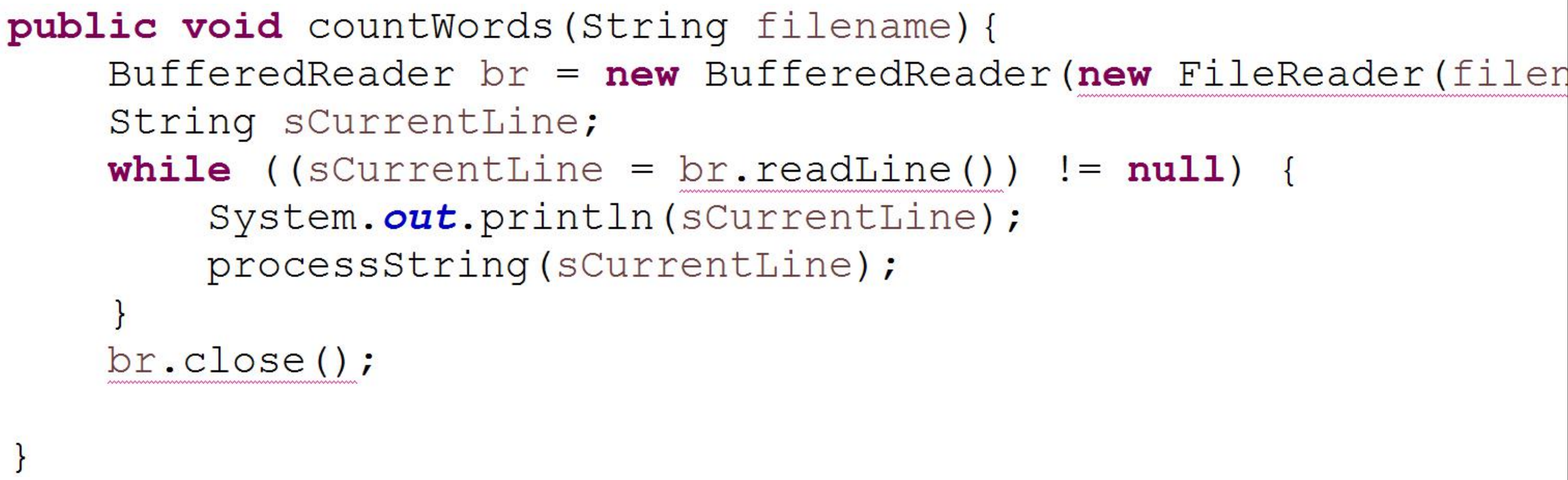
Catch Runtime Errors

```
import java.util.*;

public class HandleExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();

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

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        scanner.nextLine(); // discard input
      }
    } while (continueInput);
  }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

Example

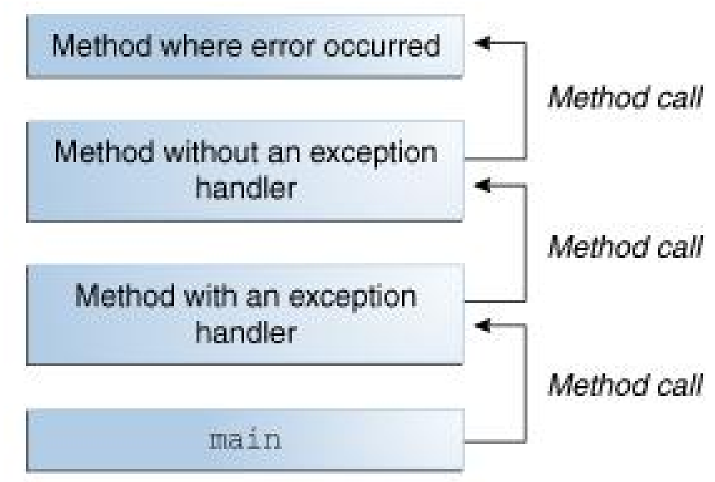
- WordCounter: counts the number of words for a given filename.



```
public void countWords(String filename){
    BufferedReader br = new BufferedReader(new FileReader(filen
    String sCurrentLine;
    while ((sCurrentLine = br.readLine()) != null) {
        System.out.println(sCurrentLine);
        processString(sCurrentLine);
    }
    br.close();

}
```

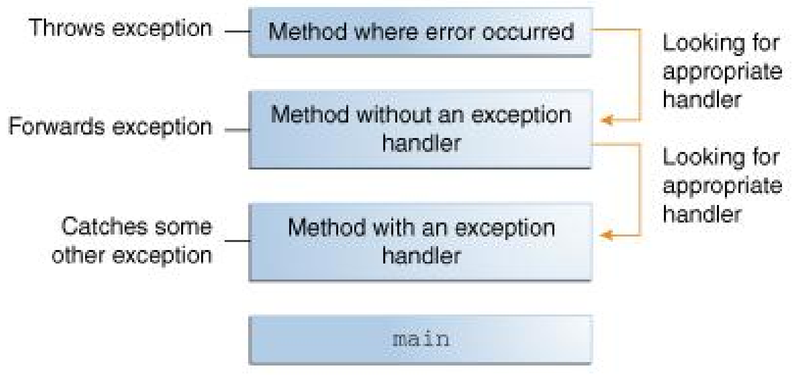
Call Stack

- After a method throws an exception, the runtime system attempts to find something to handle it.
- The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the **call stack**


Method where error occurred
Method without an exception handler
Method with an exception handler
main
Method call
Method call
Method call

Exception Handler

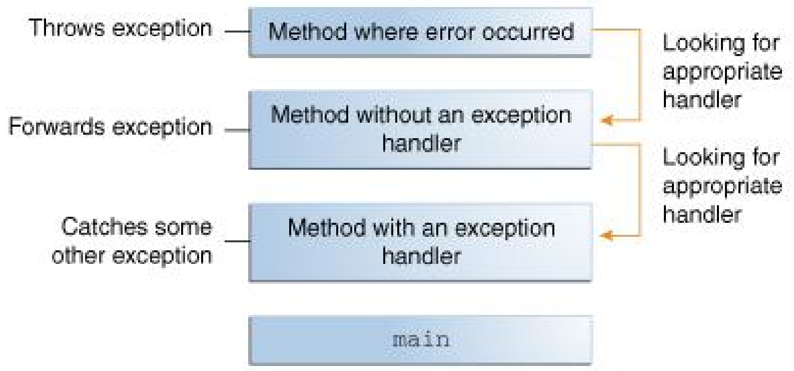
- The runtime system searches the call stack for a method that contains a block of code that can handle the exception.
- This block of code is called an exception handler.
- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

Exception Handler

- The exception handler chosen is said to catch the exception.

- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.



Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

The catch Blocks

- The catch block contains code that is executed if and when the exception handler is invoked.
- The runtime system invokes the exception handler when the handler is the first one in the call stack whose ExceptionType matches the type of the exception thrown.

Catching More Than One Type of Exception with One Exception Handler

- A single catch block can handle more than one type of exception.
  - This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Rethrowing Exceptions

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

The `finally` Block

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

The finally Block

- The finally block always executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

Next statement in the method is executed

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

Next statement;
```

animation

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

animation

Trace a Program Execution

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

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed.

animation

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.

animation

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.

animation

Trace a Program Execution

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

Next statement;
```

Handling exception

animation

Trace a Program Execution

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

Next statement;
```

Execute the final block

animation

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

When to Throw Exceptions

- An exception occurs in a method.
  - If you 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.

Chained Exceptions

- An application often responds to an exception by throwing another exception.
- The first exception causes the second exception

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

Chained Exceptions

- The following are the methods and constructors in Throwable that support chained exceptions.

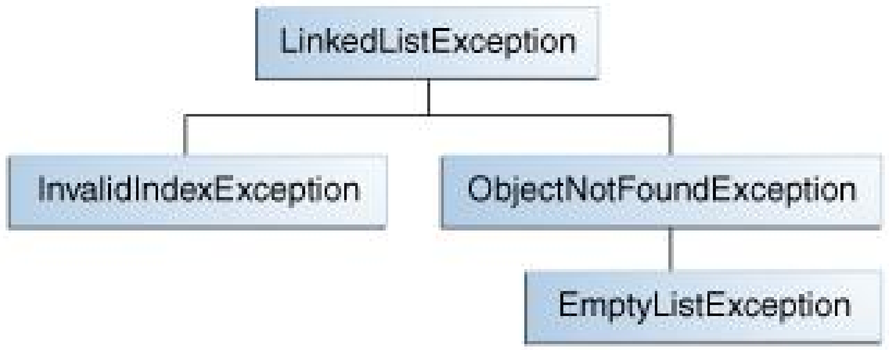
- **`Throwable getCause()`**
- **`Throwable initCause(Throwable)`**
- **`Throwable(String, Throwable)`**
- **`Throwable(Throwable)`**

Creating Exception Classes

- When faced with choosing the type of exception to throw,
  - you can either use one written by someone else (the Java platform provides a lot of exception classes you can use)
  - or you can write one of your own.

An Example: Linked List Class

- The class supports the following methods:
  - **objectAt(int n)** — Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.
  - **firstObject()** — Throws an exception if the list contains no objects.
  - **indexOf(Object o)** — Throws an exception if the object passed into the method is not in the list.



Checked vs. Runtime

- Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous.
- Having to add runtime exceptions in every method declaration would reduce a program's clarity.
- Thus, the compiler does not require that you catch or specify runtime exceptions
- In general, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Checked vs. Runtime

- Here's the bottom line guideline:
  - If a client can reasonably be expected to recover from an exception, make it a checked exception.
  - If a client cannot do anything to recover from the exception, make it an unchecked exception.

Creating Custom Exception Classes

- ☞ Use the exception classes in the API whenever possible.
- ☞ Create custom exception classes if the predefined classes are not sufficient.
- ☞ Declare custom exception classes by extending Exception or a subclass of Exception.

Advantages of Exceptions

- Advantage 1: Separating Error-Handling Code from "Regular" Code
  - consider the pseudocode method here that reads an entire file into memory.

```
readFile {

   open the file;  //What happens if the file can't be opened?

   determine its size; // What happens if the length of the file can't be determined?

   allocate that much memory; //What happens if enough memory can't be allocated?

   read the file into memory; //What happens if the read fails?

   close the file; //What happens if the file can't be closed?
}
```

Advantages of Exceptions

- **Advantage 2: Propagating Errors Up the Call Stack**
  - Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

Advantages of Exceptions

- **Advantage 2: Propagating Errors Up the Call Stack**
  - Suppose also that **method1** is the only method interested in the errors that might occur within readFile.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}
errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}
errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

Advantages of Exceptions

- **Advantage 3: Grouping or categorizing of exceptions as class hierarchy.**
  - IOException is the most general and represents any type of error that can occur when performing I/O.
  - Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.
  - A method can write specific handlers that can handle a very specific exception.

```
catch (FileNotFoundException e) {
    ...
}
```

  - A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement.

```
catch (IOException e) {
    ...
}
```

Advantages of Exceptions

- **Advantage 3: Grouping or categorizing of exceptions as class hierarchy.**
  - You could even set up an exception handler that handles any Exception with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

Questions

- Is the following code legal?

```
try {

} finally {

}
```

Questions

- What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

Questions

- Is there anything wrong with this exception handler as written? Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

References

- http://math.hws.edu/javanotes/
- http://docs.oracle.com/javase/tutorial/essential/index.html