Error Handling in Java

Exceptions

Motivation

- We seek **robust** programs

- When something unexpected occurs (an error)
  - Ensure program detects the problem

  - Then program must do something about it
    - It must specify how to handle each type of error that can happen

Traditional Methods of Handling Errors

- In most procedural languages, the standard way of indicating an error condition is by returning an error code.
- The calling code typically does one of the following:
  - Checks the error code and takes the appropriate action
  - Ignores the error code
- It was considered good programming practice to only have one entry point and one exit point from any given function.
  - This often lead to very convoluted code.
  - If an error occurred early in a function, the error condition would have to be carried through the entire function to be returned at the end. This usually involved a lot of if statements and usually lead to gratuitously complex code
  - Makes the code harder to read
    - Normal code is mixed with the code for error handling

Example

```
processFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    do some processing;
    close the file;
    return result;
}
```

Example – Mixing regular code and error handling code

```
processFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    do some processing;
    close the file;
    return result;
}
```

Check error code after each method invocation

```
errorCodeType processFile {
    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;
                do some processing
                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;
}
```

And if method is already using the return value?

Solution in Java: Exception

- Separates error handling code from the main-line code
- Made using the *Exception* concept
  - Represented by the ***Exception*** class
  - An exception means that an action member cannot complete the task it was supposed to perform as indicated by its name
  - When an *error* occurs that is represented by an *Exceptional* condition
  - Each type of *error* is represented by a distinct type of *Exception*
  - Exceptions cause the current program flow to be interrupted and transferred to a registered exception handling block.
    - This might involve unrolling the method calling stack
- Exception handling involves a well-structured goto

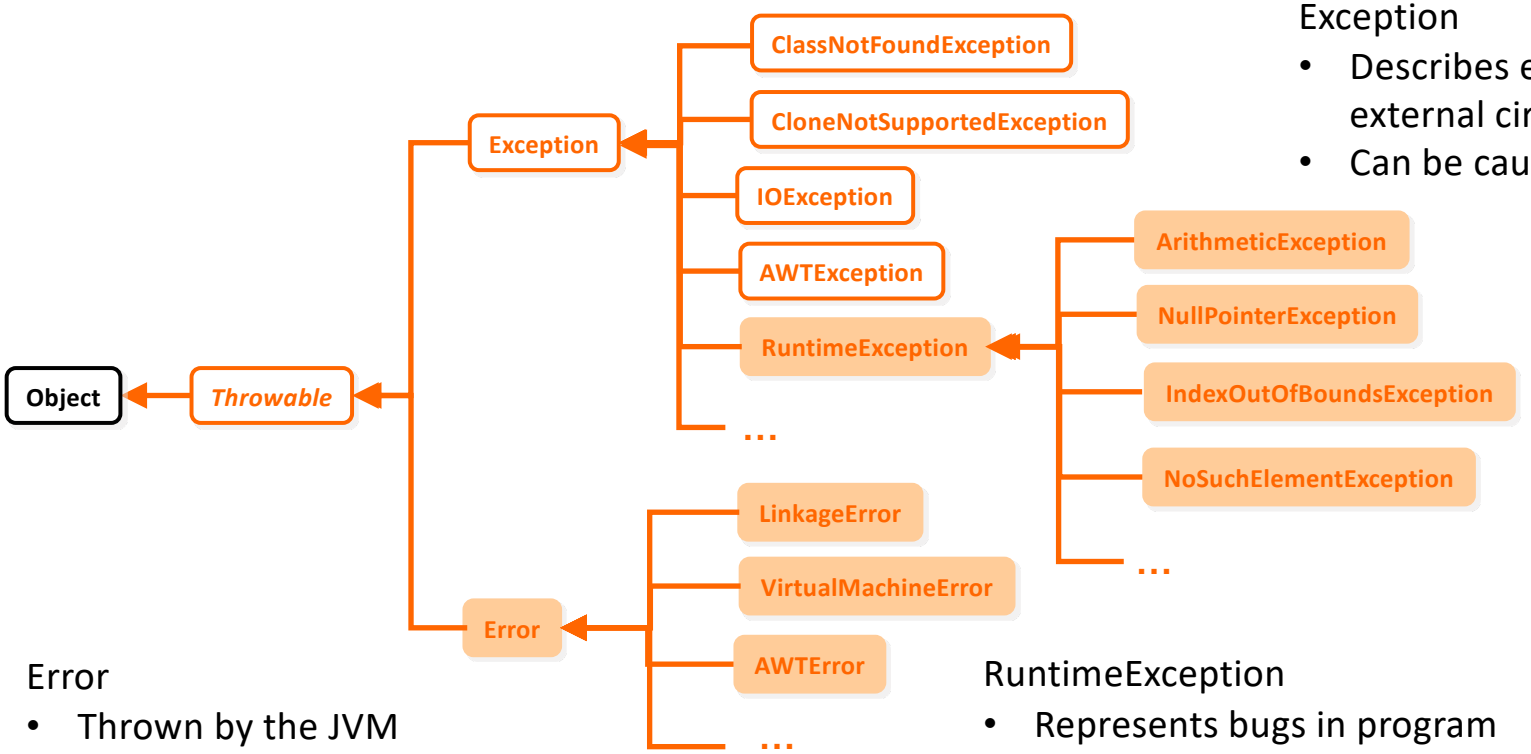
Exception - Terminology

- When an error is detected, an exception is *thrown*
- Any exception which is thrown, must be caught by an exception handler
  - If the programmer hasn't provided one, the exception will be caught by a catch-all exception handler provided by the system
  - The default exception handler terminates the execution of the application
- Exceptions can be rethrown if the exception cannot be handled by the block which caught the exception
- Java has 5 keywords for exception handling:
  - try
  - catch
  - finally
  - throw
  - throws

Checked and Unchecked Exceptions

- Java has three types of exceptions:
  - Checked
    - Represent an exception condition that an application should anticipate and recover from
    - If your code invokes a method which is defined to throw a checked exception, your code MUST provide a catch handler or declare the exception
    - The compiler generates an error if the appropriate catch handler is not present
  - Errors
    - Exceptional conditions that are external to the application and that the application usually cannot anticipate and recover from
  - Runtime exceptions
    - Exceptional conditions that are internal to the application and that the application usually cannot anticipate and recover from
    - Usually represent bug in the code
  - Unchecked
    - Errors and runtime exceptions are collectively known as unchecked exceptions
    - Not required to catch or declare
    - an unchecked exception is not caught, it will go to the default catch-all handler for the application

Java Exception Class Hierarchy



Exception
- Describes errors caused by your program and external circumstances
- Can be caught and handled by your program

Checked

Exceptions that must be caught or declared

Unchecked

Represent bugs or serious errors that a typical program should not handle

Error
- Thrown by the JVM
- Describe *system errors*
- Usually, cannot do much to recover

RuntimeException
- Represents bugs in program
- Caused by programming errors:
  - Bad casting
  - Accessing an out-of-bounds array
  - Invocation on null reference

Main methods available in the Throwable class

- **public String getMessage()**
  - Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor
- **public Throwable getCause()**
  - Returns the cause of the exception as represented by a Throwable object
- **public String toString()**
  - Returns the name of the class concatenated with the result of getMessage()
- **public void printStackTrace()**
  - Prints the result of toString() along with the stack trace to System.err

How to Handle Exceptions?

- Java **try** block is used to enclose the code that might throw an exception
  - It must be used within a method
- Each **try** block must be followed by either **catch** or **finally** block
  - Both types of blocks can be present. It is mandatory to have at least one
- A **catch** block is associated with a given exception type
  - Specifies the exception handling for that exception type
  - You can use multiple **catch** blocks for a single **try** block
  - A **catch** statement only handles exceptions that happen in the context of its try block
- **finally** block
  - This block is always executed whether an exception happened or nor
  - used to execute important code such as closing connection, streams, etc.

Example – Java version

```
processFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    do some processing;
    close the file;
    return result;
}
```

Check error code after each method invocation

```
processFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        process the file
        close the file;
    } catch (fileOpenFailed) {
        doSomething1;
    } catch (sizeDeterminationFailed) {
        doSomething2;
    } catch (memoryAllocationFailed) {
        doSomething3;
    } catch (readFailed) {
        doSomething4;
    } catch (fileCloseFailed) {
        doSomething5;
    }
```

Declaring Exceptions

- Every method must state the **checked** exceptions it might throw
  - This is known as declaring exceptions
  - **Mandatory only** for checked exceptions
  - A method can throw multiple exceptions
    - Multiple exceptions are separated by commas after the **throws** keyword

```
public class MyClass {
  public int doSomething() throws SomeException, AnotherException {
    [...]
  }
}
```

  - If a method invokes another that throws exceptions and it does not handle them, then it must declare the exceptions

```
public void method1() throws SomeException, AnotherException {
  MyClass anObject = new MyClass();
  int theSize = anObject.doSomething();
  [...]
}
```

Invoking code with checked exceptions

- Any code which throws a checked exception MUST be placed within a try block or declare the exceptions

```
public class MyClass {
  public int doSomething() throws IOException
  [...]
```

```
public void method1() {
  MyClass anObject = new MyClass();
  int theSize = anObject.doSomething();
  […]
}
```

handle

```
public void method1() {
  MyClass anObject = new MyClass();
  try {
      int theSize = anObject.doSomething();
      […]
  } catch (IOException x) {
      // ...
  }
}
```

declare

```
public void method1() throws IOException {
  MyClass anObject = new MyClass();
  int theSize = anObject.doSomething();
  […]
}
```

Catching Multiple Exceptions

- Each try block can catch multiple exceptions.
- Start with the most specific exceptions
  - *FileNotFoundException* is a subclass of *IOException*
  - It **MUST** appear before *IOException* in the catch list
  - Newer versions of Java give compilation error

```
public void method1() {
 FileInputStream aFile;
 try {
     aFile = new FileInputStream(...);
     int aChar = aFile.read();
     //...
 } catch(FileNotFoundException x) {
     // ...
 } catch(IOException x) {
       // …
 }
 // ...
```

The catch-all Handler

- Since all *Exception* classes are a subclass of the Exception class, a catch handler which catches "*Exception*" will catch all exceptions
- It must be the last in the catch List
- **NEVER** have empty catch's
  - **Hides potential bugs**

```
try {
  // execute that may throw SomeExceptionClass
} catch(SomeExceptionClass sec) {
}
```

The **finally** Block

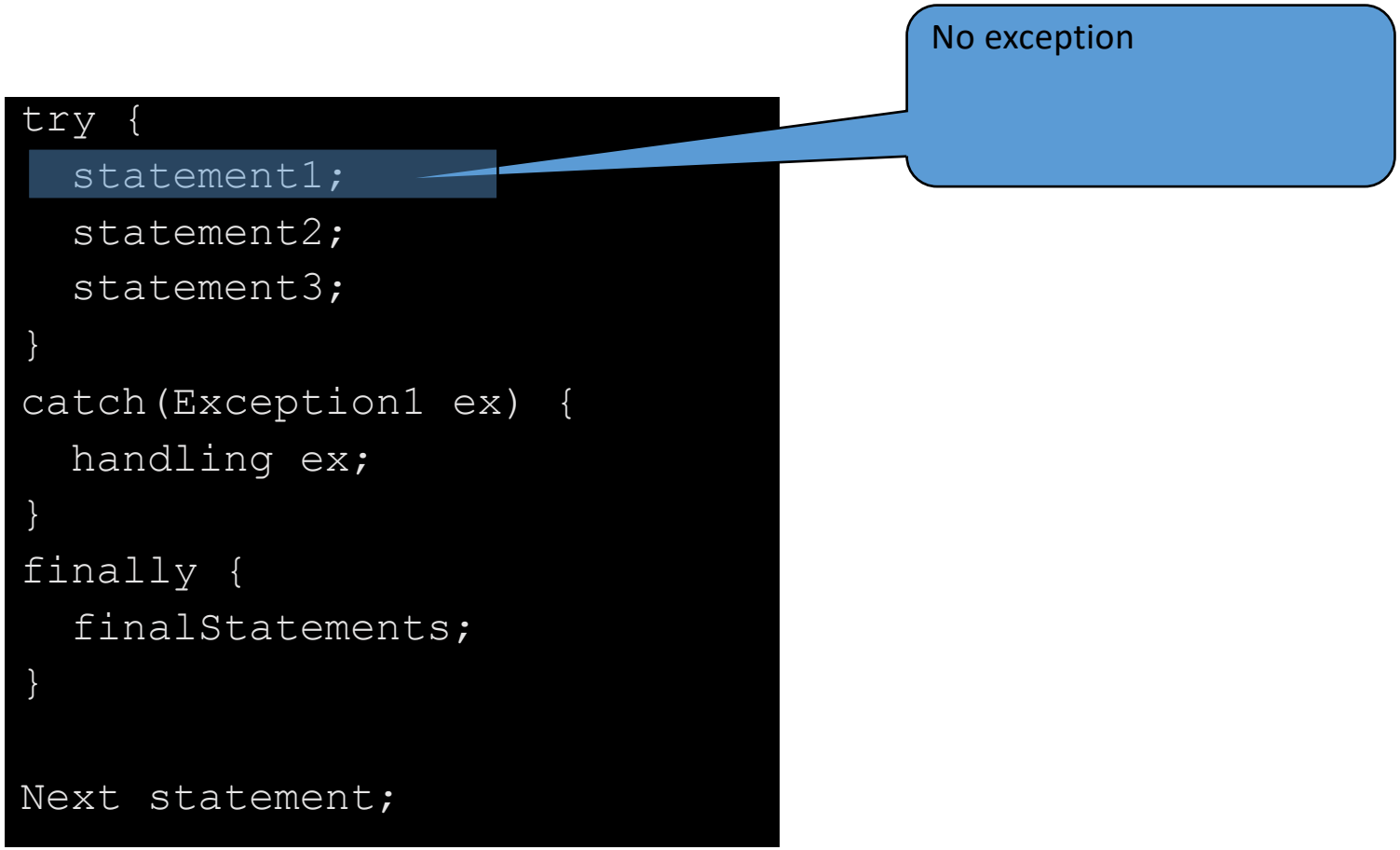
- A block that can exist for a try
- It is always executed
  - Contains last statements executed
    - Except if there is a return or throw
  - Factorizes common code in try and catch blocks

```
public void method1() {
  FileInputStream aFile = null;
  try {
    aFile = new FileInputStream(...);
    int aChar = aFile.read();
    //...
  } catch(IOException x) {
    // ...
  } catch(Exception x) {
    // Catch All other Exceptions
  } finally {
    try {
      aFile.close();
    } catch (IOException x) {
      // close might throw an exception
    }
  }
```

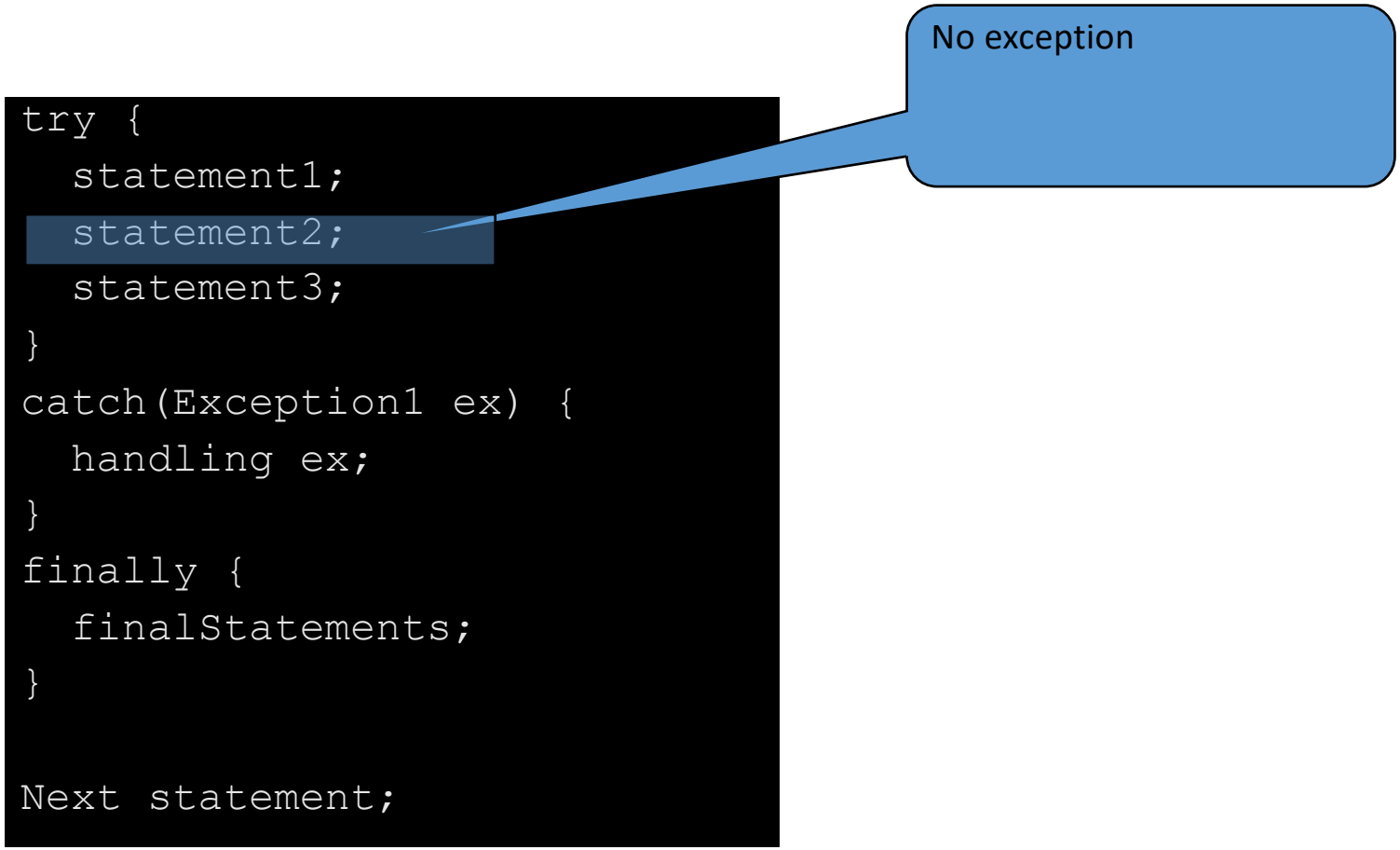
Trace a Program Execution

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

Next statement;
```

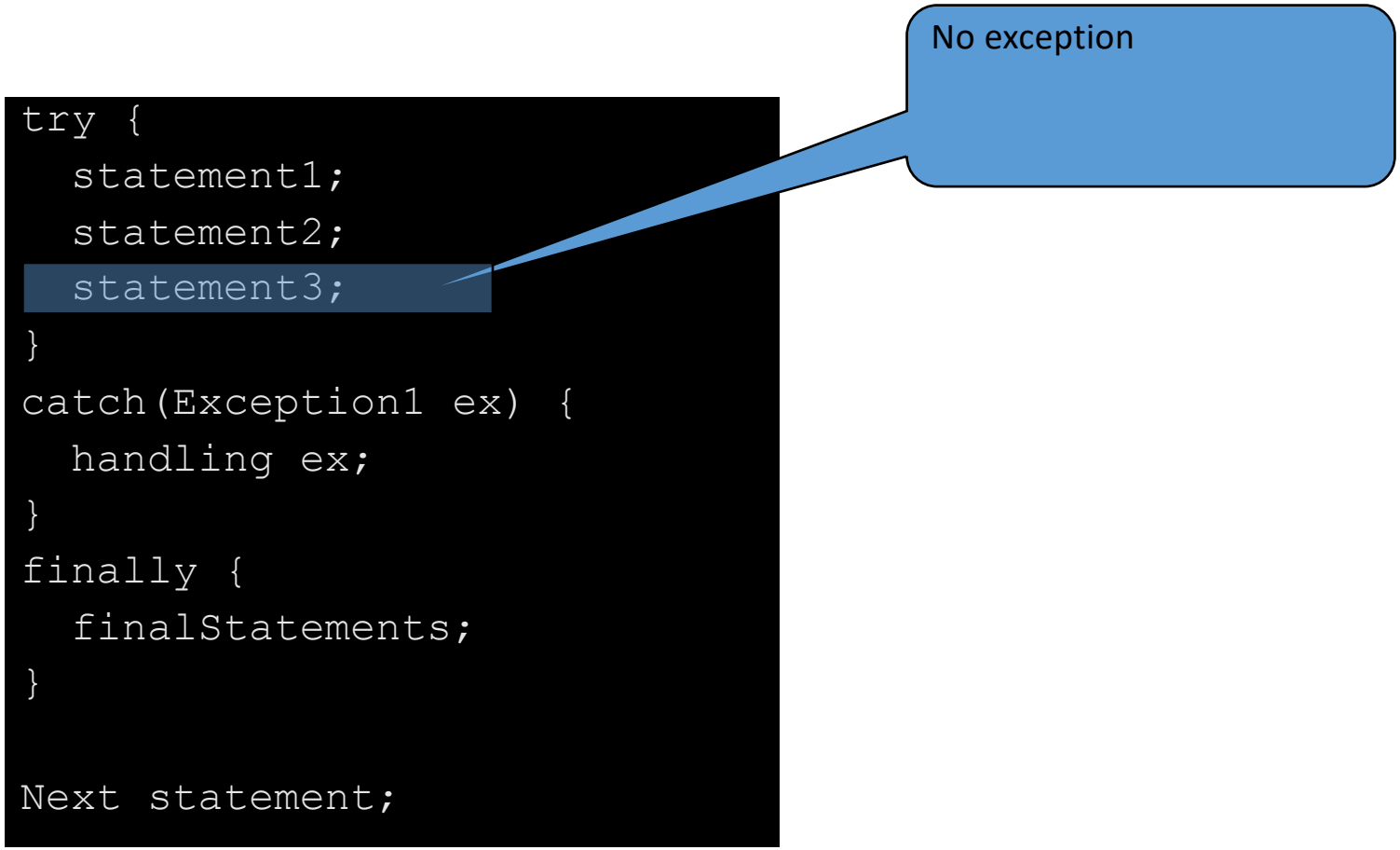
Trace a Program Execution – Without exceptions (1)



Trace a Program Execution – Without exceptions (2)

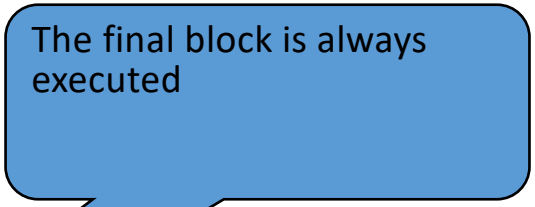


Trace a Program Execution – Without exceptions (3)



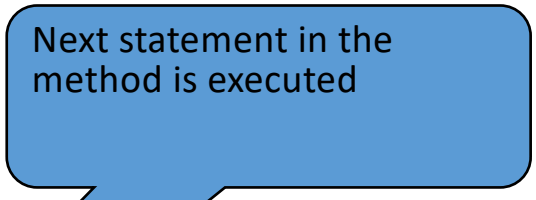
Trace a Program Execution – Without exceptions (4)

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

Next statement;
```

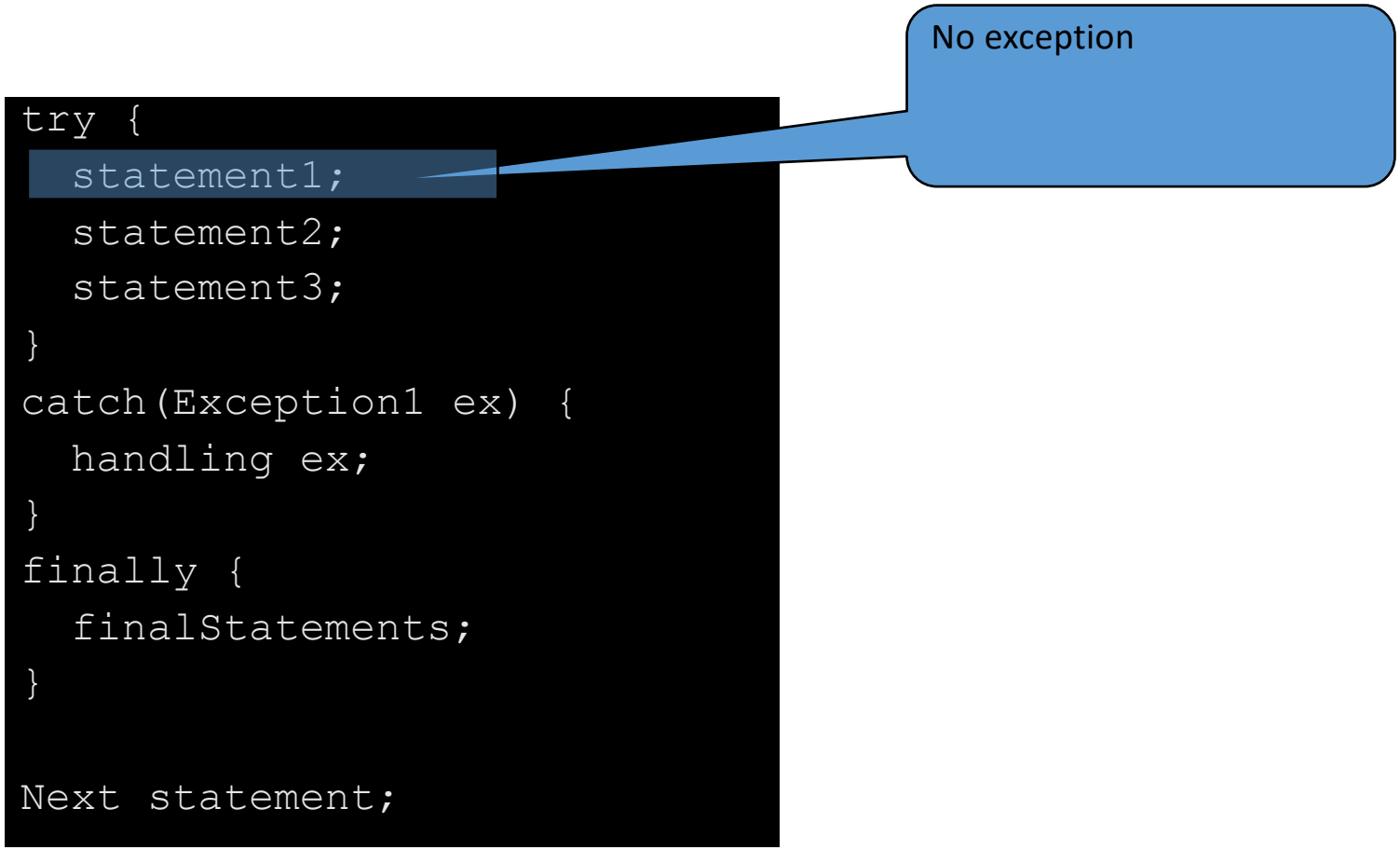


Trace a Program Execution – Without exceptions (5)

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

Next statement;
```



Trace a Program Execution – With exception in statement2 (1)



Trace a Program Execution – With exception in statement2 (2)

```
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 – With exception in statement2 (3)

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

Next statement;
```

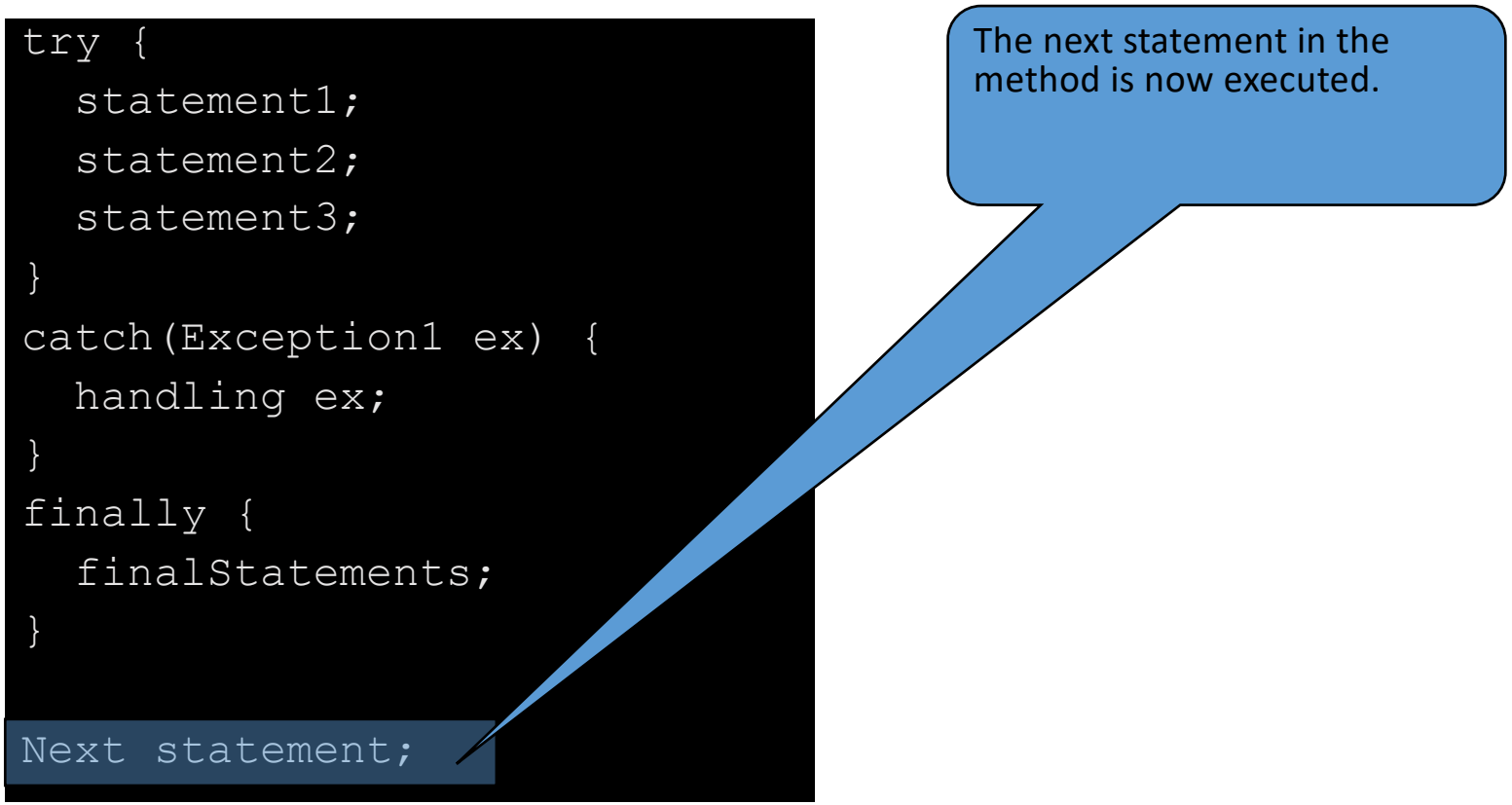
The exception is handled.

Trace a Program Execution – With exception in statement2 (4)

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

Next statement;
```

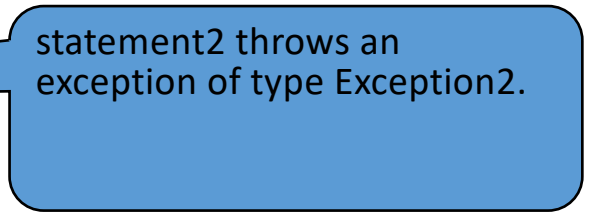
The final block is always executed.

Trace a Program Execution – With exception in statement2 (5)



Trace a Program Execution – With exception and rethrow (1)

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

Next statement;
```

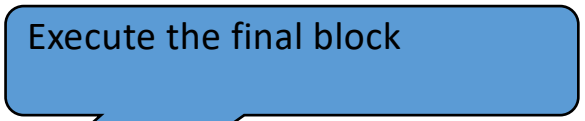


Trace a Program Execution – With exception and rethrow (2)

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

Next statement;
```

Trace a Program Execution – With exception and rethrow (3)

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

Next statement;
```



Trace a Program Execution – With exception and rethrow (4)

```
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

Similar with a ***return*** instead of ***throw***

The try-with-resources Block

- Generally, when we use any resources like streams, connections, etc. we have to close them explicitly using finally block.
- A better approach is to use the try-with-resources handling mechanism
  - Also called **automatic resource management**
- try-with-resources block
  - Declared the required resources within the parenthesis after **try**
  - The declared resources will be automatically closed at the end of the block
  - Each class used in the arguments of the try block must implement **AutoCloseable** interface
  - Each resource declared at the try block is implicitly declared as final

The try-with-resources Block - Example

Without **try-with-resources**

```
public void method1() {
  FileInputStream aFile = null;
  try {
    aFile = new FileInputStream(...)
    int aChar = aFile.read();
    //...
  } catch(IOException x) {
    // ...
  } catch(Exception x) {
    // Catch All other Exceptions
  } finally {
    try {
      aFile.close();
    } catch (IOException x) {
      // close might throw an exception
    }
  }
}
```

With **try-with-resources**

```
public void method1() {

  try (FileInputStream aFile = new FileInputStream(...)) {
       int aChar = aFile.read();
       //...
  } catch(IOException x) {
       // ...
  } catch(Exception x) {
       // Catch All other Exceptions
  }
}
```

Throwing Exceptions

- You can throw exceptions from your own methods
- To throw an exception, create an instance of the exception class and "throw" it
  - Use the throw keyword
- If you throw checked exceptions, you must indicate which exceptions your method throws by using the throws keyword
- For non-cheched exceptions you may omit them in the throws clause

Throwing Exceptions - Example

- Consider the withdraw method of *BankAccount* class

```
public void withdraw(float anAmount) throws InsufficientFundsException {
 if (anAmount < 0.0)
   throw new IllegalArgumentException(”Invalid negative amt");
 if (anAmount > balance)
    throw new InsuffientFundsException("Not enough cash");

 balance = balance - anAmount;
}
```

- Anything strange?
  - `IllegalArgumentException` thrown but not declared
  - No problem since it is an unchecked exception

Defining your own Exceptions

- To define your own exception you must do the following:
  - Create an exception class to hold the exception data
  - This exception class can specify fields to better describe the *error*
  - Your exception class must subclass "Exception" or another exception class
    - Note: to create unchecked exceptions, subclass the *RuntimeException* class
  - Minimally, your exception class should provide a constructor which takes the exception description as its argument
- To throw your own exceptions:
  - When an exceptional condition occurs, create a new instance of the exception and throw it.
  - If your exception is checked, any method which is going to throw the exception must define it using the throws keyword
    - Or catch it in a catch block but …

Invoking Code that Throws Exceptions

- For **checked** exceptions must follow the **handle or declare** rule
  - Method knows how to handle the exception
    - Specify a catch block for the specific exception type
  - Method does not know how to handle exception
    - Declare exception in throws clause of the method
- Can change the exception or re-throw the exception in catch block
  - Change exception
    - Create a new exception, using a more appropriate exception for the upper invoking levels, and throw this exception in catch block
    - Designates as chained exception
  - Re-throw exception
    - Catch an exception object and throw the same exception object, using the throw keyword

Catching Exceptions – Correct the Example

```
void method1() {
  method2();
}
```

```
void method2() {
  try {
    …
   method3();
  } catch (Exception3 ex) {
   System.err.println("Error!");
  }  catch (Exception4 ex) {
   if (condition)
     throw new Exception2();
   else
     throw ex;
  }
}
```

```
void method3() {
  …
  if (condition)
    throw new Exception3();
  method4();
}
```

Correct

```
void method4() {
  throw new Exception4();
}
```

Catching Exceptions – Correct the Example

```
void method1() {
  method2();
}
```

```
void method2() {
  try {
    …
   method3();
  } catch (Exception3 ex) {
   System.err.println(“Error!”);
  }  catch (Exception4 ex) {
   if (condition)
     throw new Exception2();
   else
     throw ex;
  }
}
```

Correct

```
void method3() {
  …
  if (condition)
    throw new Exception3();
  method4();
}
```

```
void method4()  throws Exception4 {
  throw new Exception4();
}
```

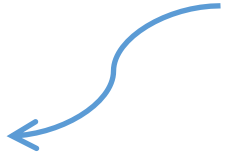
Catching Exceptions – Correct the Example

```
void method1() {
  method2();
}
```

```
void method3() throws Exception3, Exception4 {
  …
  if (condition)
    throw new Exception3();
  method4();
}
```

```
void method2() {
  try {
    …
   method3();
  } catch (Exception3 ex) {
   System.err.println("Error!");
  }  catch (Exception4 ex) {
   if (condition)
     throw new Exception2();
   else
     throw ex;
  }
}
```

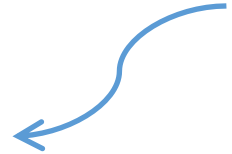
Correct



```
void method4()  throws Exception4 {
  throw new Exception4();
}
```

Catching Exceptions – Correct the Example

Correct



```
void method1() {
  method2();
}
```

```
void method3() throws Exception3, Exception4 {
  …
  if (condition)
    throw new Exception3();
  method4();
}
```

```
void method2() throw Exception2, Exception4 {
  try {
    …
   method3();
  } catch (Exception3 ex) {
   System.err.println("Error!");
  }  catch (Exception4 ex) {
   if (condition)
     throw new Exception2();
   else
     throw ex;
  }
}
```

```
void method4()  throws Exception4 {
  throw new Exception4();
}
```

Catching Exceptions – Correct the Example

```
void method1() throws Exception2, Exception4 {
  method2();
}
```

```
void method2() throws Exception2, Exception4 {
  try {
    …
   method3();
  } catch (Exception3 ex) {
   System.err.println("Error!");
  }  catch (Exception4 ex) {
   if (condition)
     throw new Exception2();
   else
     throw ex;
  }
}
```

```
void method3() throws Exception3, Exception4 {
  …
  if (condition)
    throw new Exception3();
  method4();
}
```

```
void method4()  throws Exception4 {
  throw new Exception4();
}
```

Chained Exceptions

- Map one exception type to another
  - This way, a method can throw exceptions defined at the same abstraction level as the method itself
  - Hides implementation details of invoked method
- One exception causes to throw other exception
  - The first one is the cause of the second one
  - Usually, it is good to know the cause of the second one
    - More information available for handling exception

Available Constructors in Exception

- public Exception()
  - Cause and message are not initialized
- public Exception(String message)
  - Initializes message. Cause is not initialized
- public Exception(String message, Throwable cause)
  - Constructs a new exception with the specified detail message and cause
- public Exception(Throwable cause)
  - Constructs a new exception with the specified cause and a detail message of (cause==null ? null : cause.toString())

When to Use Exceptions?

- When a method does not know how to handle an exceptional situation
  - Exception should be handled by one of its callers
- Do not use try-catch block to deal with simple, expected situations

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

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

Custom Exceptions - Example

- **InsufficientFundsException**

```
public class InsufficientFundsException extends Exception {
  private double amount;

  public InsufficientFundsException(double amount) {
    super("Not enough funds. Amount is to big. " + amount);
    this.amount = amount;
  }

  public double getAmount() {
    return amount;
  }
}
```

- Chained Exception **- CustomException**

```
public class CustomException extends Exception {
  private String _someField;

  public CustomException(String str) {
    _someField = str;
  }

  public CustomException(String message, String str) {
    super(message);
    _someField = str;
  }

  public CustomException(String message, Throwable cause, String str) {
    super(message, cause);
    _someField = str;
  }

  public String getSomeField() {
    return _someField;
  }
}
```

Exception Handling Best Practices

- Single catch block for multiple exceptions
  - Group together all catch blocks that are similar
- Use Specific Exceptions
  - Base classes of *Exception* hierarchy do not provide any useful information
  - Use Custom Exceptions
- Throw early, Catch late
  - It is better to declare that a method throws a checked exception than to handle the exception poorly
- Naming Conventions
  - Exception classes should end with *Exception*
- Close resources
- Exceptions should not be used for controlling the control flow of programs
  - Exceptions should represent abnormal conditions

```
try {
   …
} catch (ArrayIndexOutOfBoundsException e) {
  System.err.println(”Error: ” + e.getMessage());
} } catch (NullPointerEXception e) {
  System.err.println(”Error: ” + e.getMessage());
}
```

```
try {
   …
} catch (ArrayIndexOutOfBoundsException |
            NullPointerEXception e) {
  System.err.println(”Error: ” + e.getMessage());
}
```

```
try {
  for (int i = 0;; i++) {
    System.out.println(args[i]);
  }
} catch (ArrayIndexOutOfBoundsException e) {
  // do something
}
```

Nested try Statements

- ✓A **try** statement can be inside the block of another try
- ✓Each time a **try** statement is entered, the context of that exception is pushed on the stack
- ✓If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match