哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

Chapter 7: Software Construction for Robustness

7.2 Error and Exception Handling

Ming Liu

April 20, 2018

Outline

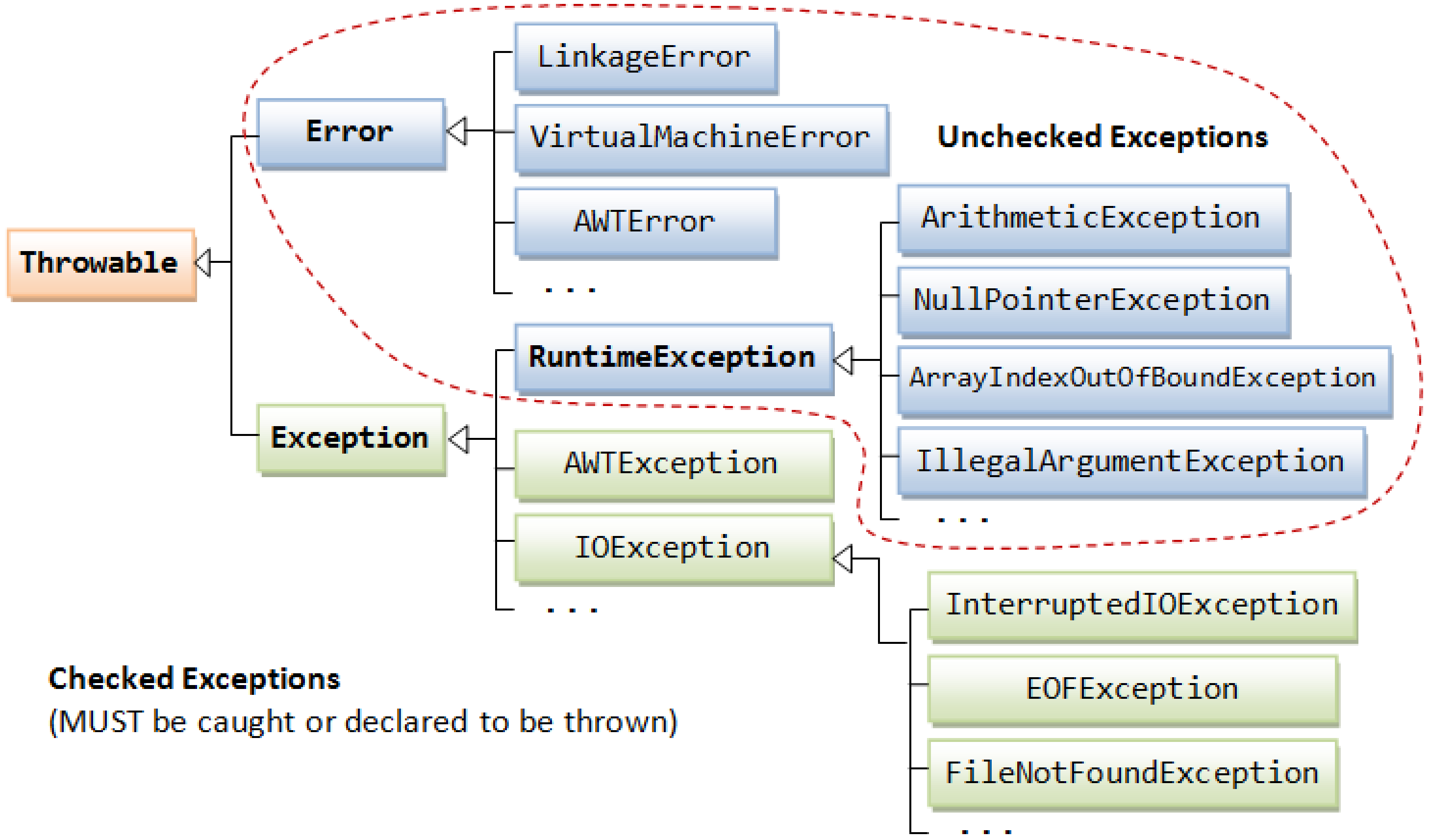
- **1 Error and Exception in Java**
- **2 Error Handling Techniques**
- **3 Exception Handling**
  - What is Exception?
  - Classification of exceptions
  - Checked and unchecked exceptions
  - How to Throw an Exception
  - Creating Exception Classes
  - Catching Exceptions
  - Rethrowing and Chaining Exceptions
  - finally Clause
  - The Try-with-Resources Statement
  - Analyzing Stack Trace Elements
  - Tips for Using Exceptions
- **4 Summary**

HARBIN INSTITUTE OF TECHNOLOGY

1 Error and Exception in Java

"Abnormals" in Java

- **The base class for all Exception objects is `java.lang.Throwable`, together with its two subclasses `java.lang.Exception` and `java.lang.Error`.**



Error and Exception

- **The `Error` class describes internal system errors and resource exhaustion situations inside the Java runtime system (e.g., `VirtualMachineError`, `LinkageError`) that rarely occur.**
  - You should not throw an object of this type.
  - There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully.

- **The `Exception` class describes the error caused by your program (e.g. `FileNotFoundException`, `IOException`).**
  - These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

2 Error Handling Techniques

Sorts of errors

- **User input errors**
  - In addition to the inevitable typos, some users like to blaze their own trail instead of following directions. E.g. a user asks to connect to a URL that is syntactically wrong, the network layer will complain.
- **Device errors**
  - Hardware does not always do what you want it to. The printer may be turned off. A web page may be temporarily unavailable. Devices will often fail in the middle of a task.
- **Physical limitations**
  - Disks can fill up; you can run out of available memory
- **Code errors**
  - A method may not perform correctly. e.g. it could deliver wrong answers or use other methods incorrectly. Computing an invalid array index, trying to find a nonexistent entry in a hash table, or trying to pop an empty stack are all examples of a code error.

Error handling

- **Error handling refers to the anticipation, detection, and resolution of programming, application, and communications errors.**
- **Specialized programs, called error handlers, are available to deal with errors.**

- **Possible strategies of error handling:**
  - Forestall errors if possible
  - Gracefully terminate an affected application and save the error information to a log file

Dealing with Errors

- **While encountering errors, or an operation cannot be completed because of an error, the program ought to either**
  - Notify the user of an error
  - Save all work
  - Return to a safe state and enable the user to execute other commands
  - Allow the user to save all work and terminate the program gracefully
- **This may not be easy to do**, because the code that detects (or even causes) the error condition is usually far removed from the code that can roll back the data to a safe state or the code that can save the user’s work and exit cheerfully.
- **The mission of error handling** is to transfer control from where the error occurred to an error handler that can deal with the situation.

(1) Return a neutral value

- **Return a neutral value (Robustness)**
  - Sometimes the best response to bad data is to continue operating and simply return a value that's known to be harmless.
  - E.g., A numeric computation might return 0. A string operation might return an empty string, or a pointer operation might return an empty pointer. A drawing routine that gets a bad input value for color might use the default background or foreground color.
- **However, for a drawing program that shows a patient's X-ray film, it is best not to display a neutral value.**
- **In this case, it's better to close the program than to show the wrong patient data. (Correctness, not robustness)**

(2) Substitute the next piece of valid data

- **Substitute the next piece of valid data (Robustness)**
  - When processing a stream of data, some circumstances call for simply returning the next valid data.
  - E.g., If you're reading records from a database and encounter a corrupted record, you might simply continue reading until you find a valid record. If you're taking readings from a thermometer 100 times per second and you don't get a valid reading one time, you might simply wait another 1/100th of a second and take the next reading.

(3) Log a warning message to a file

- **Log a warning message to a file (Robustness)**
  - When bad data is detected, you might choose to log a warning message to a file and then continue on.
  - This approach can be used in conjunction with other techniques like substituting the closest legal value or substituting the next piece of valid data.

(4) Return the same answer as the previous time

- **Return the same answer as the previous time (Robustness)**
  - If a thermometer-reading software doesn't get a reading one time, it might simply return the same value as last time. Depending on the application, temperatures might not be very likely to change much in 1/100th of a second. In a video game, if you detect a request to paint part of the screen an invalid color, you might simply return the same color used previously.
- **However, if you are managing a deal on an ATM, you may not want to use this rule, because that is the previous user's bank account. (Correctness)**

(5) Substitute the closest legal value

- **Substitute the closest legal value (Robustness)**
  - In some cases, you might choose to return the closest legal value.
  - This is often a reasonable approach when taking readings from a calibrated instrument. The thermometer might be calibrated between 0 and 100 degrees Celsius, for example. If you detect a reading less than 0, you can substitute 0 which is the closest legal value. If you detect a value greater than 100, you can substitute 100.
  - For a string operation, if a string length is reported to be less than 0, you could substitute 0.

(6) Return an error code

- **Return an error code (Correctness)**
  - You could decide that only certain parts of a system will handle errors; other parts will not handle errors locally; they will simply report that an error has been detected and trust that some other routine higher up in the calling hierarchy will handle the error.
  - The specific mechanism for notifying the rest of the system that an error has occurred could be any of the following:
    - Set the value of a status variable
    - Return status as the function's return value
    - Throw an exception using the language's built-in exception mechanism
  - In this case, the specific error-reporting mechanism is less important than the decision about which parts of the system will handle errors directly and which will just report that they've occurred. If security is an issue, be sure that calling routines always check return codes.

(7) Call an error processing routine/object

- **Call an error processing routine/object (Correctness)**
  - This is to centralize error handling in a global error handling routine or error handling object.
  - The advantage is that error processing responsibility can be centralized, which can make debugging easier.
  - The tradeoff is that the whole program will know about this central capability and will be coupled to it. If you ever want to reuse any of the code from the system in another system, you'll have to drag the error handling machinery along with the code you reuse.

- **This approach has an important security implication.**
  - If your code has encountered a buffer-overrun, it's possible that an attacker has compromised the address of the handler routine or object.
  - Thus, once a buffer overrun has occurred while an application is running, it is no longer safe to use this approach.

(8) Display an error message

- **Display an error message wherever the error is encountered (Correctness)**
  - This approach minimizes error-handling overhead, however it does have the potential to spread user interface messages through the entire application, which can create challenges when you need to create a consistent user interface, try to clearly separate the UI from the rest of the system, or try to localize the software into a different language.
  - Also, beware of telling a potential attacker of the system too much. Attackers sometimes use error messages to discover how to attack a system.

(9) Handle the error locally

- **Handle the error in whatever way works best locally (Correctness)**
  - Some designs call for handling all errors locally—the decision of which specific error-handling method to use is left up to the programmer designing and implementing the part of the system that encounters the error.
  - This approach provides individual developers with great flexibility, but it creates a significant risk that the overall performance of the system will not satisfy its requirements for correctness or robustness.
  - Depending on how developers end up handling specific errors, this approach also has the potential to spread user interface code throughout the system, which exposes the program to all the problems associated with displaying error messages.

(10) Shutdown

- **Shutdown (Correctness)**
  - Some systems shut down whenever they detect an error.
  - This approach is useful in safety critical applications.
    - For example, if the software that controls radiation equipment for treating cancer patients receives bad input data for the radiation dosage, shutting down is the best option.
    - We'd much prefer to reboot the machine than to run the risk of delivering the wrong dosage.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

3 Exception Handling

An Example

```
public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}

public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds");
    }
}
```

(1) What is Exception?

Exceptions

- **An exception is an *abnormal event* that arises during the execution of the program and disrupts the normal flow of the program.**

- **Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it.**
- **Java allows every method an alternative exit path if it is unable to complete its task in the normal way**.
  - In this situation, the method ***throws*** an object that encapsulates the error information. The method exits immediately and does not return any value. Moreover, execution does not resume at the code that called the method;
  - Instead, the ***exception-handling*** mechanism begins its search for an exception handler that can deal with this particular error condition.

Benefits of exceptions

- **You can't forget to handle common failure modes**
  - Compare: using a flag or special return value
- **Provide high-level summary of error, and stack trace**
  - Compare: core dump in C
- **Improve code structure**
  - Separate normal code path from exceptional
  - Ease task of recovering from failure
- **Ease task of writing robust, maintainable code**

Benefits of exceptions

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
   switch (errno) {
     case _ENOFILE:
         System.err.println("File not found: " + …);
         return -1;
     default:
         System.err.println("Something else bad happened: " + …);
         return -1;
   }
}

DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
   System.err.println("Unknown internal error.");
   return -1; // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
   System.err.println("Error reading binary data from file");
   return -1;
}
return i;
```

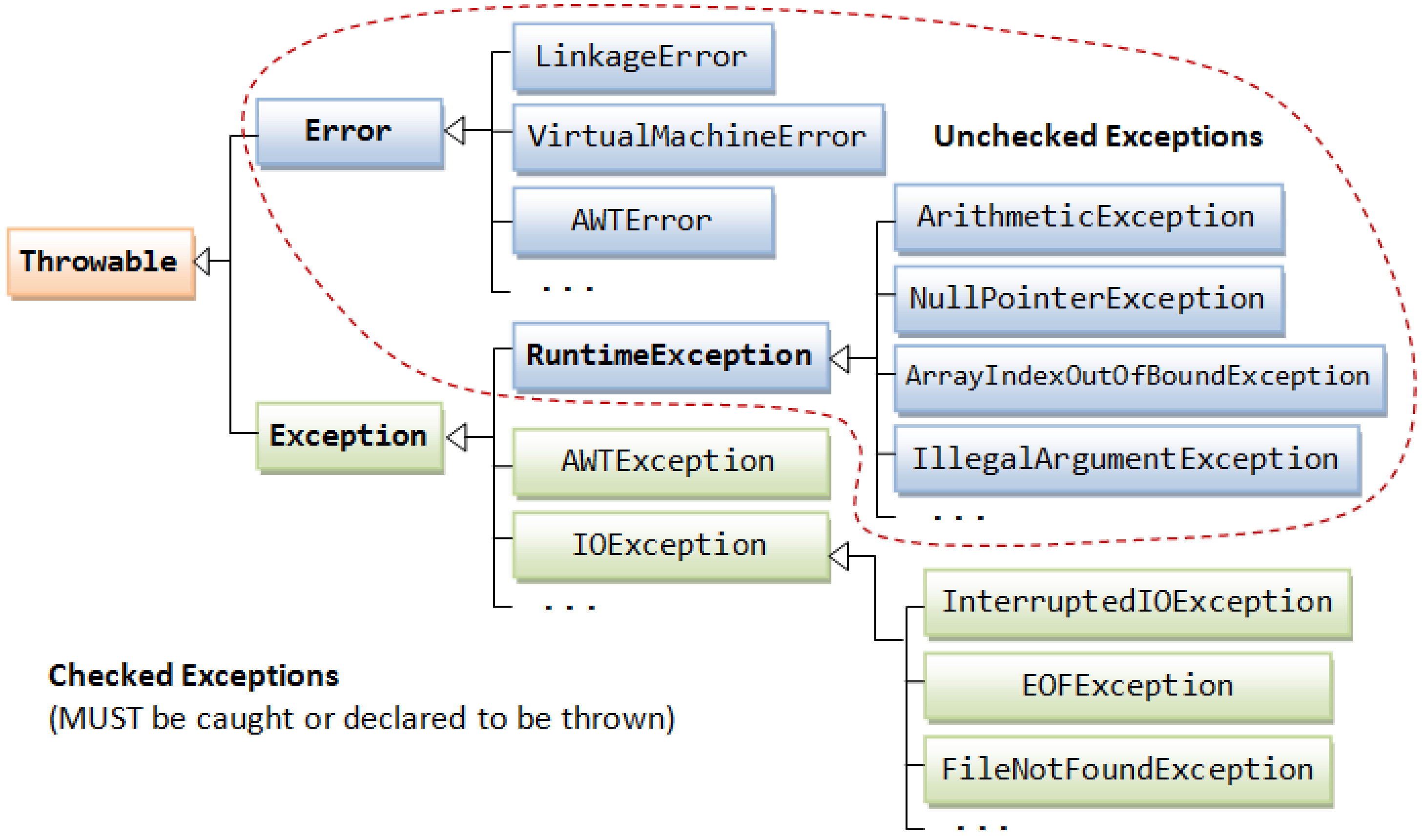
Benefits of exceptions

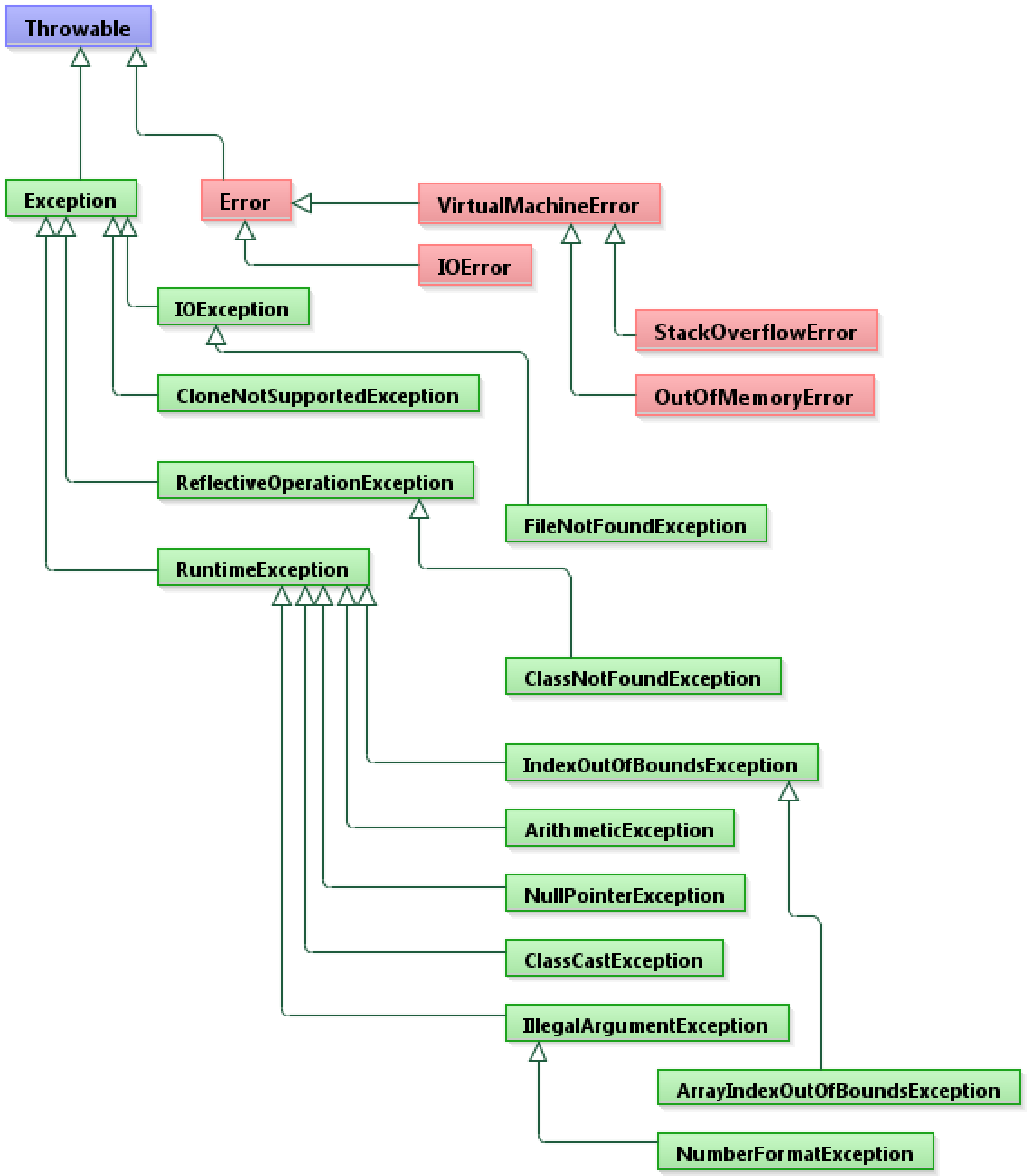
```
FileInputStream fileInput = null;

try {
        fileInput = new FileInputStream(fileName);
        DataInput dataInput = new DataInputStream(fileInput);
        return dataInput.readInt();
}
catch (FileNotFoundException e) {
        System.out.println("Could not open file " + fileName);
}
catch (IOException e) {
        System.out.println("Couldn’t read file: " + e);
}
finally {
        if (fileInput != null) fileInput.close();
}
```

(2) Classification of exceptions

Exceptions are derived from `Throwable`

- **In Java programming language, an exception object is always an instance of a class derived from `Throwable`.**




Throwable
Exception
Error
VirtualMachineError
IOError
IOException
StackOverflowError
CloneNotSupportedException
OutOfMemoryError
ReflectiveOperationException
FileNotFoundException
RuntimeException
ClassNotFoundException
IndexOutOfBoundsException
ArithmeticException
NullPointerException
ClassCastException
IllegalArgumentException
ArrayIndexOutOfBoundsException
NumberFormatException

Runtime Exception and Other Exceptions

- **When doing Java programming, focus on the *Exception* hierarchy.**
- **The *Exception* hierarchy also splits into two branches: Exceptions that derive from `RuntimeException` and those that do not.**
- **General rule:**
  - A `RuntimeException` happens because you made a programming error.
  - Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.

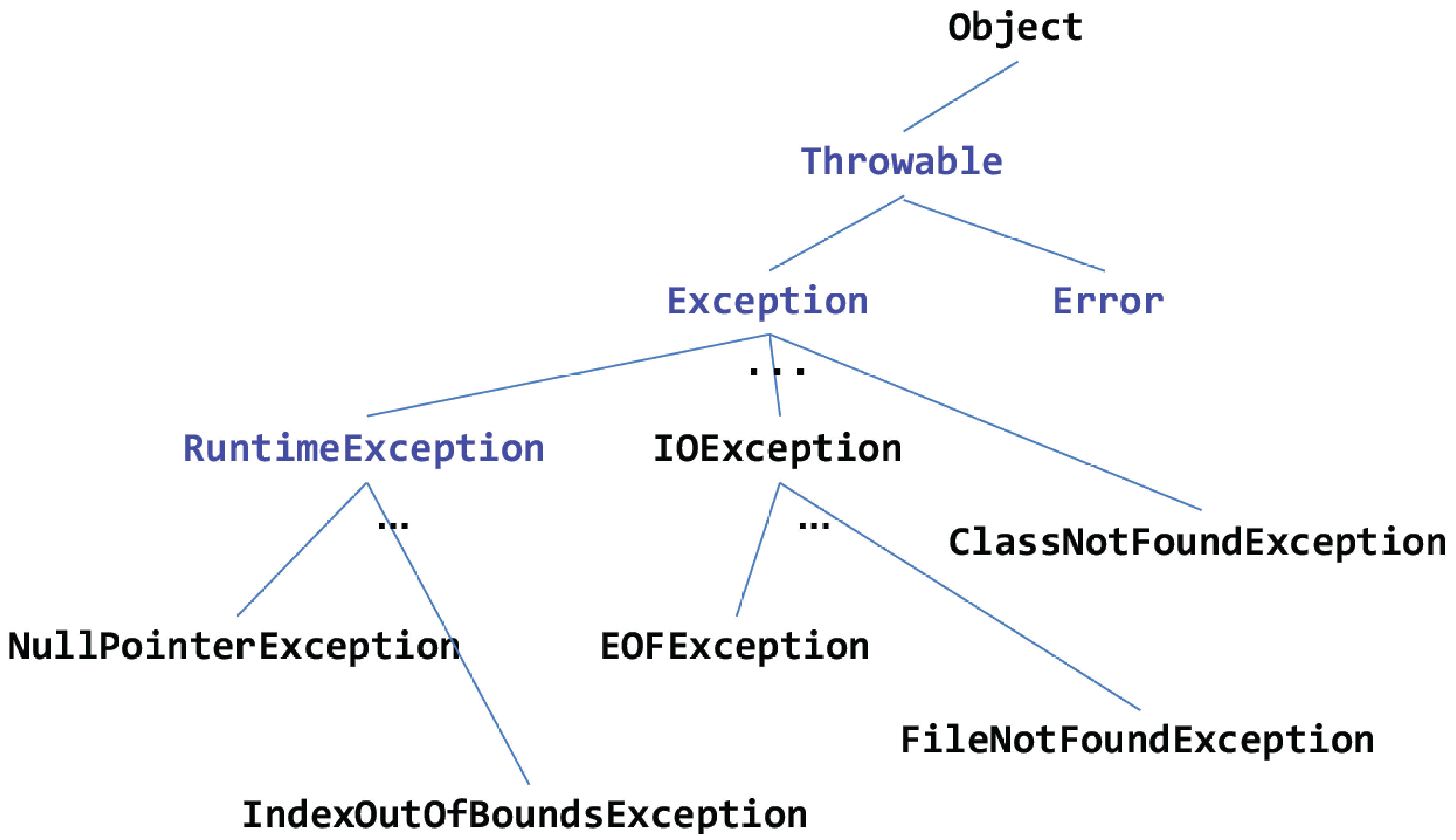
`RuntimeException`

- **If it is a `RuntimeException`, it was your fault**
  - You could have avoided `ArrayIndexOutOfBoundsException` by testing the array index against the array bounds.
  - The `NullPointerException` would not have happened had you checked whether the variable was null before using it.

- **How about a file that doesn't exist?**
  - Can't you first check whether the file exists, and then open it?
  - Actually, the file might be deleted right after you checked for its existence. Thus, the notion of "existence" depends on the environment, not just on your code.

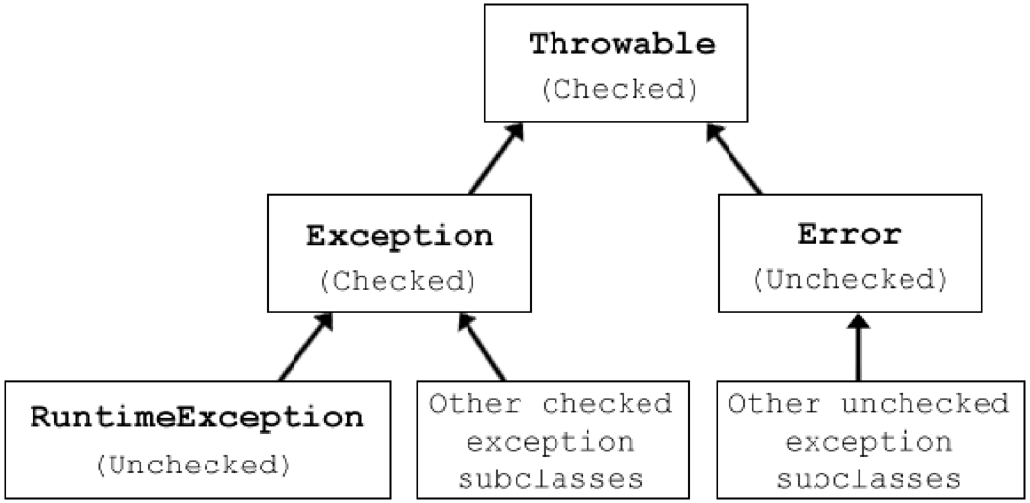
(3) *Checked* and *unchecked* exceptions

Again, the exception hierarchy in Java



Checked/unchecked exceptions

- **Unchecked exception: Programming error, other unrecoverable failure (Error + RuntimeException)**
  - No action is required for program to compile, but uncaught exception will cause program to fail
- **Checked exception: An error that every caller should be aware of and handle**
  - Must be caught or propagated, or program won't compile (The compiler checks that you provide exception handlers for all checked exceptions)



Common Unchecked Exception Classes

- `ArrayIndexOutOfBoundsException`: **thrown by JVM when your code uses an array index, which is outside the array's bounds.**

```
int[] anArray = new int[3];
System.out.println(anArray[3]);
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

- `NullPointerException`: **thrown by the JVM when your code attempts to use a null reference where an object reference is required.**

```
String[] strs = new String[3];
System.out.println(strs[0].length());
Exception in thread "main" java.lang.NullPointerException
```

Common Unchecked Exception Classes

- `NumberFormatException`: **Thrown programmatically (e.g., by Integer.parseInt()) when an attempt is made to convert a string to a numeric type, but the string does not have the appropriate format.**

```
Integer.parseInt("abc");
```

```
Exception in thread "main" java.lang.NumberFormatException:
For input string: "abc"
```

- `ClassCastException`: **thrown by JVM when an attempt is made to cast an object reference fails.**

```
Object o = new Object();
Integer i = (Integer)o;
```

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Object cannot be cast to java.lang.Integer
```

Common Unchecked Exception Classes

- `IllegalArgumentException`: thrown programmatically to indicate that a method has been passed an illegal or inappropriate argument. You could re-use this exception for your own methods.
- `IllegalStateException`: thrown programmatically when a method is invoked and the program is not in an appropriate state for that method to perform its task. This typically happens when a method is invoked out of sequence, or perhaps a method is only allowed to be invoked once and an attempt is made to invoke it again.
- `NoClassDefFoundError`: thrown by the JVM or class loader when the definition of a class cannot be found.

Checked Exception Handling Operations

- **Five keywords are used in exception handling**: `try, catch, finally, throws` **and** `throw`.

- **Java's exception handling consists of three operations:**
  - Declaring exceptions (`throws`)
  - Throwing an exception (`throw`)
  - Catching an exception (`try, catch, finally`)

Declaring Checked Exceptions by `throws`

- **A Java method can throw an exception if it encounters a situation it cannot handle.**
  - A method will not only tell the Java compiler what values it can return, it is also going to tell the compiler what can go wrong.
  - E.g. code that attempts to read from a file knows that the file might not exist or that it might be empty. The code that tries to process the information in a file therefore will need to notify the compiler that it can throw some sort of `IOException`.
- **The place in which you advertise that your method can throw an exception is the header of the method;** the header changes to reflect the checked exceptions the method can throw.

```
public FileInputStream(String name)
    throws FileNotFoundException
```

Declaring Checked Exceptions by `throws`

- **When you write your own methods, you don't have to advertise every possible `throwable` object that your method might actually throw.**
- **To understand when (and what) you have to advertise in the *throws clause* of the methods you write, keep in mind that an exception is thrown in any of the following four situations:**
  - You call a method that throws a checked exception—for example, the `FileInputStream` constructor.
  - You detect an error and throw a checked exception with the `throw` statement.
  - You make a programming error, such as `a[-1] = 0` that gives rise to an unchecked exception (`ArrayIndexOutOfBoundsException`).
  - An internal error occurs in the virtual machine or runtime library.

Declare more than one Checked Exceptions

- **If a method might throw more than one checked exception type, you must list all exception classes in the header. Separate them by commas, as in the following example**

```
class MyAnimation {
    . . .
    public Image loadImage(String s)
                throws FileNotFoundException, EOFException
    {
      . . .
    }
}
```

Don’t throw `Error` and unchecked exceptions

- **Do not need to advertise internal Java errors—that is, exceptions inheriting from `Error`.**
  - Any code could potentially throw those exceptions, and they are entirely beyond your control.
- **Similarly, you should not advertise unchecked exceptions inheriting from `RuntimeException`.**
  - These runtime errors are completely under your control.
  - If you are so concerned about array index errors, you should spend your time fixing them instead of advertising the possibility that they can happen.

```
void drawImage(int i) throws ArrayIndexOutOfBoundsException
// bad style
```

Summary

- **Summary:**
  - A method must declare all the *checked* exceptions that it might throw.
  - If your method fails to faithfully declare all checked exceptions, the compiler will issue an error message.
  - Unchecked exceptions are either beyond your control (Error) or result from conditions that you should not have allowed in the first place (RuntimeException).
- **Caution:**
  - If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method.
  - It is OK to throw more specific exceptions, or not to throw any exceptions in the subclass method.
  - In particular, if the superclass method throws no checked exception at all, neither can the subclass.

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(4) How to Throw an Exception

How to Throw an Exception

- **Suppose you have a method, `readData`, that is reading in a file. Something terrible has happened in your code.**
- **You may decide this situation is so abnormal that you want to throw an exception `EOFException` with the description "Signals that an EOF has been reached unexpectedly during input."**

```
throw new EOFException();
```

or

```
EOFException e = new EOFException();
throw e;
```

How to Throw an Exception

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (. . .)
    {
        if (!in.hasNext()) // EOF encountered
        {
            if (n < len)
                 throw new EOFException();
         }
          . . .
     }
      return s;
}
```

How to Throw an Exception

- **The `EOFException` has a second constructor that takes a `string` argument.**
- **You can put this to good use by describing the exceptional condition more carefully.**

```
String gripe = "Content-length: " + len + ", Received: " + n;

throw new EOFException(gripe);
```

How to Throw an Exception

- **Throwing an exception is easy if one of the existing exception classes works for you:**
  - Find an appropriate exception class.
  - Make an object of that class.
  - Throw it.

- **Once a method throws an exception, it does not return to its caller. This means you do not have to worry about cooking up a default return value or an error code.**

(5) Creating Exception Classes

Creating Exception Classes

- **Your code may run into a problem which is not adequately described by any of the standard exception classes.**
- **In this case, it is easy enough to create your own exception class.**

- **Just derive it from `Exception`, or from a child class of `Exception` such as `IOException`.**
- **It is customary to give both a default constructor and a constructor that contains a detailed message.**
  - The `toString` method of the `Throwable` superclass returns a string containing that detailed message, which is handy for debugging.

Creating Exception Classes

```
class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}
```

```
String readData(BufferedReader in) throws FileFormatException {
    . . .
    while (. . .){
        if (ch == -1) { // EOF encountered
            if (n < len)
                throw new FileFormatException();
        }
        . . .
    }
    return s;
}
```

(6) Catching Exceptions

Catching Exceptions

- **If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace.**
- **GUI programs catch exceptions, print stack trace messages, and then go back to the user interface processing loop.**
- **To catch an exception, set up a `try/catch` block :**

```
try {
    code
    more code
    more code
}
catch (ExceptionType e) {
    handler for this type
}
```

Catching Exceptions

- **If any code inside the `try` block throws an exception of the class specified in the catch clause, then**
  - The program skips the remainder of the code in the ***try block***.
  - The program executes the handler code inside the ***catch clause***.

  - If none of the code inside the `try` block throws an exception, then the program skips ***the catch clause***.
  - If any of the code in a method throws an exception of a type other than the one named in the ***catch clause***, this method exits immediately. (Hopefully, one of its callers has already provided a catch clause for that type.)

Catching Exceptions

```
public void read(String filename) {

    try {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1) {
            process input...
        }
    }
    catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

Pass the exception on to the caller

- **Another choice to handle the exceptions: do nothing at all and simply pass the exception on to the caller.**
  - Let the caller of the read method worry about it!
  - Might be the best choice (prefer more correctness to robustness).
- **If we take that approach, then we have to advertise the fact that the method may throw an `IOException`.**

```
public void read(String filename) throws IOException {
     InputStream in = new FileInputStream(filename);
     int b;
     while ((b = in.read()) != -1) {
          process input
     }
}
```

- **The compiler strictly enforces the `throws` specifiers. If you call a method that throws a checked exception, you must either handle it or pass it on.**

Try/catch and throw an exception?

- **As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle.**
- **When you propagate an exception, you must add a throws specifier to alert the caller that an exception may be thrown.**
- **Notice:**
  - If you are writing a method that overrides a superclass method which throws no exceptions, then you must catch each checked exception in the method's code.
  - You are not allowed to add more throws specifiers to a subclass method than are present in the superclass method.

Get detailed information from an Exception

- **The exception object may contain information about the nature of the exception.**
  - To find out more about the object, try `e.getMessage()` to get the detailed error message (if there is one)
  - Use `e.getClass().getName()` to get the actual type of the exception object.

Catching Multiple Exceptions

- **You can catch multiple exception types in a try block and handle each type differently.**
- **Use a separate catch clause for each type as in the following example:**

```
try {
     code that might throw exceptions
}
catch (FileNotFoundException e) {
     emergency action for missing files
}
catch (UnknownHostException e) {
     emergency action for unknown hosts
}
catch (IOException e) {
     emergency action for all other I/O problems
}
```

(7) Rethrowing and Chaining Exceptions

Rethrowing and Chaining Exceptions

- **You can throw an exception in a `catch` clause.**
- **Typically, you do this when you want to change the exception type. If you build a subsystem that other programmers use, it makes a lot of sense to use an exception type that indicates a failure of the subsystem.**
  - E.g., `ServletException`, the code that executes a servlet may not want to know in minute detail what went wrong, but it definitely wants to know that the servlet was at fault.

Rethrowing and Chaining Exceptions

- **Here is how you can catch an exception and rethrow it:**

```
try {
     access the database
}
catch (SQLException e) {
     throw new ServletException("database error: " + e.getMessage());
}
```

- **Here, the `ServletException` is constructed with the message text of the exception.**

Rethrowing and Chaining Exceptions

- **However, it is a better idea to set the original exception as the "cause" of the new exception:**

```
try {
     access the database
}
catch (SQLException e) {
     Throwable se = new ServletException("database error");
     se.initCause(e);
     throw se;
}
```

- **When the exception is caught, the original exception can be retrieved**

```
Throwable e = se.getCause();
```

- **This wrapping technique is highly recommended. It allows you to throw high level exceptions in subsystems without losing the details of the original failure.**

(8) `finally` Clause

`finally` Clause

- **When your code throws an exception, it stops processing the remaining code in your method and exits the method.**
- **This is a problem if the method has acquired some resource (files, database connections,…), which only this method knows about, and that resource must be cleaned up.**

- **One solution is to catch and rethrow all exceptions.**
- **But this solution is tedious because you need to clean up the resource allocation in two places—in the normal code and in the exception code.**

- **Java has a better solution:** ***the finally clause.***

Try-Catch-Finally

- **The code in the finally clause executes whether or not an exception was caught.**
- **In the following example, the program will close the file *under all circumstances*:**

```
InputStream in = new FileInputStream(. . .);
try {
      // 1
      code that might throw exceptions
      // 2
}
catch (IOException e) {
      // 3
      show error message
      // 4
}
Finally {
      // 5
      in.close();
}
// 6
```

`Try-Catch-Finally`: case 1

- **Look at the three possible situations in which the program will execute the finally clause.**
- **Case 1: The code throws no exceptions.**
  - In this case, the program first executes all the code in the `try` block.
  - Then, it executes the code in the `finally` clause.
  - Afterwards, execution continues with the first statement after the `finally` clause.
  - In other words, execution passes through points 1, 2, 5, and 6.

```
InputStream in = new FileInputStream(. . .);
try {
      // 1
      code that might throw exceptions
      // 2
}
catch (IOException e) {
      // 3
      show error message
      // 4
}
Finally {
      // 5
      in.close();
}
// 6
```

`Try-Catch-Finally`: case 2

- **Case 2: The code throws an exception that is caught in a catch clause—in our case, an `IOException`.**
  - The program executes all code in the `try` block, up to the point at which the exception was thrown. The remaining code in the `try` block is skipped. The program then executes the code in the matching `catch` clause, and then the code in the `finally` clause
  - If the `catch` clause does not throw an exception, the program executes the first line after the `finally` claus
  - Execution passes through points 1, 3, 4, 5, and 6.
  - If the `catch` clause throws an exception, then the exception is thrown back to the caller of this method, and execution passes through points 1, 3, and 5 only.

```
InputStream in = new FileInputStream(. . .);
try {
      // 1
      code that might throw exceptions
      // 2
}
catch (IOException e) {
      // 3
      show error message
      // 4
}
Finally {
      // 5
      in.close();
}
// 6
```

`Try-Catch-Finally`: case 3

- **Case3: The code throws an exception that is not caught in any catch clause.**
  - Here, the program executes all code in the `try` block until the exception is thrown.
  - The remaining code in the `try` block is skipped.
  - Then, the code in the `finally` clause is executed, and the exception is thrown back to the caller of this method.
  - Execution passes through points 1 and 5 only.

```
InputStream in = new FileInputStream(. . .);
try {
      // 1
      code that might throw exceptions
      // 2
}
catch (IOException e) {
      // 3
      show error message
      // 4
}
Finally {
      // 5
      in.close();
}
// 6
```

Using `finally` without a `Catch`

- **You can use the `finally` clause without a `catch` clause.**
- **For example, consider the following `try` statement:**

```
InputStream in = . . .;
try {
     code that might throw exceptions
}
finally {
     in.close();
}
```

- **The `in.close()` statement in the `finally` clause is executed whether or not an exception is encountered in the try block.**
- **If an exception is encountered, it is rethrown and must be caught in another catch clause.**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(9) The Try-with-Resources Statement

The Try-with-Resources Statement

- **Java SE 7 provides a useful shortcut to the code pattern provided the resource belongs to a class that implements the `AutoCloseable` interface.**

```
open a resource
try {
     work with the resource
}
finally {
     close the resource
}
```

- **That interface has a single method**

```
void close() throws Exception
```

The Try-with-Resources Statement

- **In its simplest variant, the try-with-resources statement has the form**

```
try (Resource res = . . .) {
     work with res
}
```

- **When `try` block exits, then `res.close()` is called automatically.**
- **Here is a typical example—reading all words of a file**

```
try (Scanner in = new Scanner(new FileInputStream("/dict/words")), "UTF-8")
{
     while (in.hasNext())
          System.out.println(in.next());
}
```

- **When the block exits normally, or when there was an exception, the `in.close()` method is called, exactly as if you had used a `finally` block.**

The Try-with-Resources Statement

- **Specify multiple resources:**

```
try (Scanner in = new Scanner(new
     FileInputStream("/usr/share/dict/words"), "UTF-8");
     PrintWriter out = new PrintWriter("out.txt"))
{
     while (in.hasNext())
          out.println(in.next().toUpperCase());
}
```

- **No matter how the block exits, both `in` and `out` are closed.**
- **If you programmed this by hand, you would need two nested `try/finally` statements.**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(10) Analyzing Stack Trace Elements

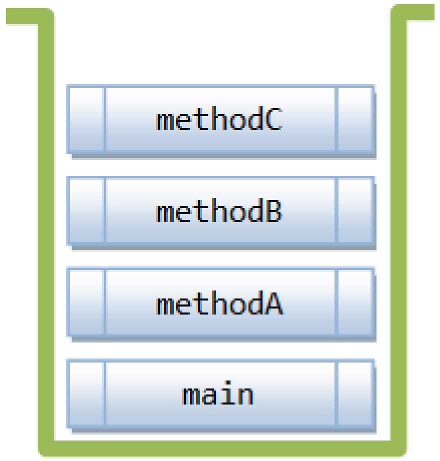
Method Call Stack

- **A typical application involves many levels of method calls, which is managed by a so-called method call stack.**
- **A stack is a last-in-first-out queue.**
- **What's the result?**
  - Enter main()
  - Enter methodA()
  - Enter methodB()
  - Enter methodC()
  - Exit methodC()
  - Exit methodB()
  - Exit methodA()
  - Exit main()

```
public class MethodCallStackDemo {
   public static void main(String[] args) {
      System.out.println("Enter main()");
      methodA();
      System.out.println("Exit main()");
   }

   public static void methodA() {
      System.out.println("Enter methodA()");
      methodB();
      System.out.println("Exit methodA()");
   }

   public static void methodB() {
      System.out.println("Enter methodB()");
      methodC();
      System.out.println("Exit methodB()");
   }

   public static void methodC() {
      System.out.println("Enter methodC()");
      System.out.println("Exit methodC()");
   }
}
```

Method Call Stack

- **The sequence of events:**
  - JVM invoke the `main()`.
  - `main()` pushed onto call stack, before invoking `methodA()`.
  - `methodA()` pushed onto call stack, before invoking `methodB()`.
  - `methodB()` pushed onto call stack, before invoking `methodC()`.
  - `methodC()` completes.
  - `methodB()` popped out from call stack and completes.
  - `methodA()` popped out from the call stack and completes.
  - `main()` popped out from the call stack and completes. Program exits.



Method Call Stack
(Last-in-First-out Queue)

Call Stack Trace

- **Suppose `methodC()` carries out a "divide-by-0" operation, which triggers an `ArithmeticException`.**
- **The exception message clearly shows the *method call stack trace* with the relevant statement line numbers:**

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

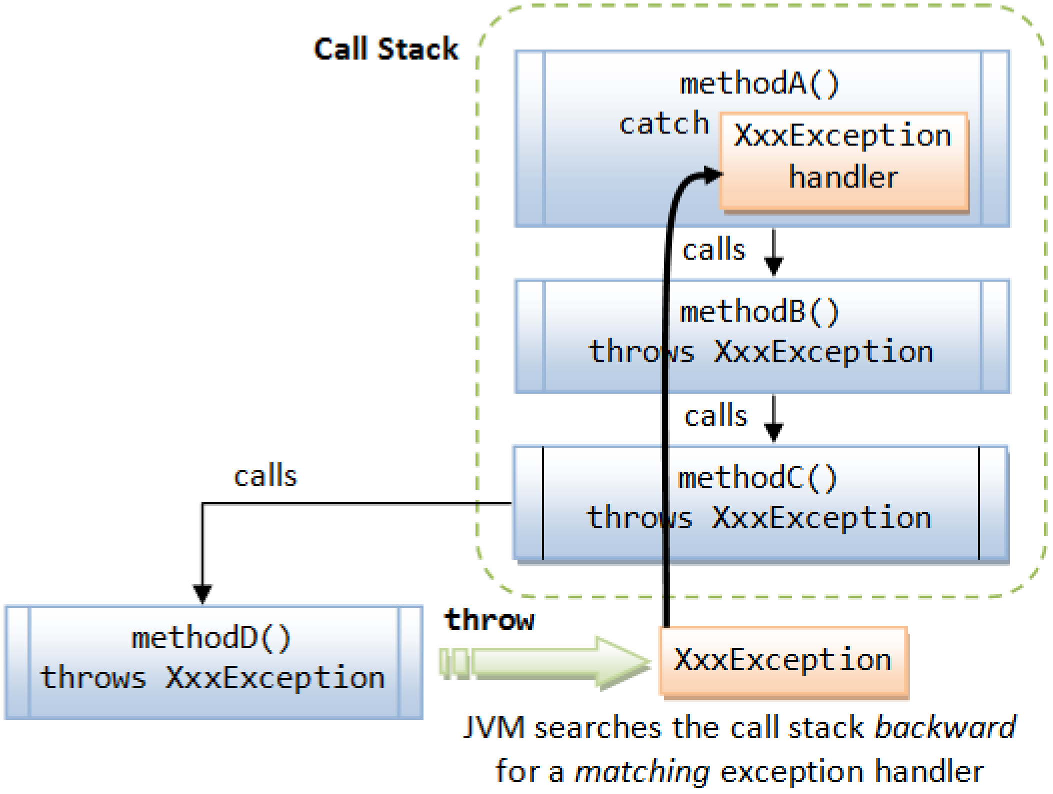
Call Stack Trace

- **Process:**
  - `MethodC()` triggers an `ArithmeticException`. As it does not handle this exception, it popped off from the call stack immediately.
  - `MethodB()` also does not handle this exception and popped off the call stack. So does `methodA()` and `main()` method.
  - The `main()` method passes back to JVM, which abruptly terminates the program and print the call stack trace.

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

Exception & Call Stack

- **Suppose that `methodD()` encounters an abnormal condition and throws a `XxxException` to the JVM.**
- **The JVM searches backward through the call stack for a matching exception handler.**
- **It finds `methodA()` having a `XxxException` handler and passes the exception object to the handler.**
  - Notice that `methodC()` and `methodB()` are required to declare "`throws XxxException`" in their method signatures in order to compile the program.

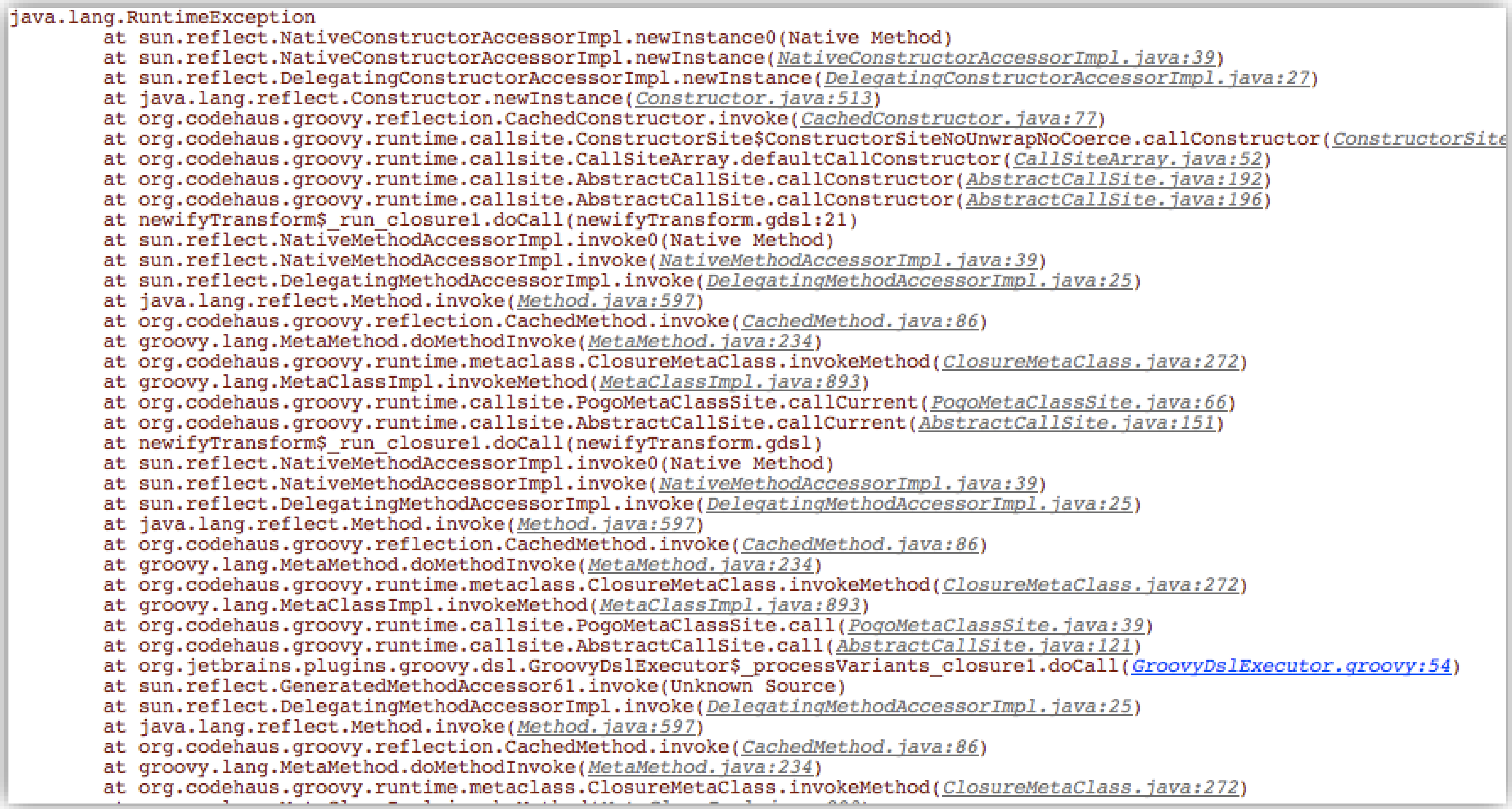


Analyzing Stack Trace Elements

- A *stack trace* is a listing of all pending method calls at a particular point in the execution of a program.
- You have almost certainly seen stack trace listings—they are displayed whenever a Java program terminates with an uncaught exception.
- You can access the text description of a stack trace by calling the `printStackTrace` method of the `Throwable` class.

```
Throwable t = new Throwable();
StringWriter out = new StringWriter();
t.printStackTrace(new PrintWriter(out));
String description = out.toString();
```

Stack trace



```
java.lang.RuntimeException
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:39)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27)
        at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
        at org.codehaus.groovy.reflection.CachedConstructor.invoke(CachedConstructor.java:77)
        at org.codehaus.groovy.runtime.callsite.ConstructorSite$ConstructorSiteNoUnwrapNoCoerce.callConstructor(ConstructorSite
        at org.codehaus.groovy.runtime.callsite.CallSiteArray.defaultCallConstructor(CallSiteArray.java:52)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:192)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callConstructor(AbstractCallSite.java:196)
        at newifyTransform$_run_closure1.doCall(newifyTransform.gdsl:21)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
        at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:893)
        at org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.callCurrent(PogoMetaClassSite.java:66)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.callCurrent(AbstractCallSite.java:151)
        at newifyTransform$_run_closure1.doCall(newifyTransform.gdsl)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
        at groovy.lang.MetaClassImpl.invokeMethod(MetaClassImpl.java:893)
        at org.codehaus.groovy.runtime.callsite.PogoMetaClassSite.call(PogoMetaClassSite.java:39)
        at org.codehaus.groovy.runtime.callsite.AbstractCallSite.call(AbstractCallSite.java:121)
        at org.jetbrains.plugins.groovy.dsl.GroovyDslExecutor$_processVariants_closure1.doCall(GroovyDslExecutor.groovy:54)
        at sun.reflect.GeneratedMethodAccessor61.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at org.codehaus.groovy.reflection.CachedMethod.invoke(CachedMethod.java:86)
        at groovy.lang.MetaMethod.doMethodInvoke(MetaMethod.java:234)
        at org.codehaus.groovy.runtime.metaclass.ClosureMetaClass.invokeMethod(ClosureMetaClass.java:272)
```

Analyzing Stack Trace Elements

- **A more flexible approach is the `getStackTrace` method that yields an array of `StackTraceElement` objects, which you can analyze in your program.**

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
    analyze frame
```

- **The `StackTraceElement` class has methods to obtain the file name and line number, as well as the class and method name, of the executing line of code.**
- **The `toString` method yields a formatted string containing all of this information.**

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

(11) Tips for Using Exceptions

Exception handling cannot replace test

- ***Exception handling is not supposed to replace a simple test.***
  - As an example, we wrote some code that tries 10,000,000 times to pop an empty stack. It first does this by finding out whether the stack is empty.

```
if (!s.empty()) s.pop();
```

  - Next, we force it to pop the stack no matter what and then catch the `EmptyStackException` that tells us we should not have done that.

```
try {
     s.pop();
}
catch (EmptyStackException e) { }
```

- The version that calls `isEmpty` ran in 646 milliseconds. The version that catches the `EmptyStackException` ran in 21,739 milliseconds --- it took far longer to catch an exception than to perform a simple test.
- The moral is: **Use exceptions for exceptional circumstances only.**

Do not micromanage exceptions

- *Do not micromanage exceptions.*
  - Many programmers wrap every statement in a separate ***try block***.
  - Next, we force it to pop the stack no matter what and then catch the `EmptyStackException` that tells us we should not have done that.

```
PrintStream out;
Stack s;
for (i = 0; i < 100; i++) {
      try {
            n = s.pop();
      }
      catch (EmptyStackException e) {
            // stack was empty
      }
      try {
            out.writeInt(n);
      }
      catch (IOException e) {
            // problem writing to
file
      }
}
```

```
try {
     for (i = 0; i < 100; i++) {
            n = s.pop();
            out.writeInt(n);
      }
}
catch (IOException e) {
      // problem writing to file
}
catch (EmptyStackException e) {
      // stack was empty
}
```

Make good use of the exception hierarchy

- *Make good use of the exception hierarchy.*
  - Don't just throw a `RuntimeException`. Find an appropriate subclass or create your own.
  - Don't just catch `Throwable`. It makes your code hard to read and maintain.
  - Do not hesitate to turn an exception into another exception that is more appropriate. For example, when you parse an integer in a file, catch the `NumberFormatException` and turn it into a subclass of `IOException` or `MySubsystemException`.

“Tough love” works better than indulgence

- ***When you detect an error, “tough love” works better than indulgence.***
  - Some programmers worry about throwing exceptions when they detect errors.
  - Maybe it would be better to return a dummy value rather than throw an exception when a method is called with invalid parameters? For example , should `Stack.pop` return null, or throw an exception when a stack is empty?
  - We think it is better to throw a `EmptyStackException` at the point of failure than to have a `NullPointerException` occur at later time.

Propagating exceptions is not a sign of shame

- ***Propagating exceptions is not a sign of shame.***
  - Many programmers feel compelled to catch all exceptions that are thrown.
  - If they call a method that throws an exception, such as the `FileInputStream` constructor or the `readLine` method, they instinctively catch the exception that may be generated.
  - Often, it is actually better to propagate the exception instead of catching it:

```
public void readStuff(String filename) throws IOException
                                            // not a sign of shame!
{
    InputStream in = new FileInputStream(filename);
    . . .
}
```

  - Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.
- **NOTE:** "throw early, catch late."

Other Tips for Using Exceptions

- **Avoid unnecessary checked exceptions**
- **Favor standard exceptions**
  - `IllegalArgumentException` – invalid parameter value
  - `IllegalStateException` – invalid object state
  - `NullPointerException` – null paramwhere prohibited
  - `IndexOutOfBoundsException` – invalid index param
- **Document all exceptions thrown by each method**
  - Checked and unchecked
  - But don't *declare* unchecked exceptions!

Other Tips for Using Exceptions

- **Include failure-capture info in detail message**

```
throw new IlegalArgumentException("Modulus must be prime: " + modulus);
```

- **Don’t ignore exceptions**

```
// Empty catch block IGNORES exception – Bad smell in code!
try {
     ...
}
catch (SomeException e) { }
```

4 Summary

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

The end

April 20, 2018