Object Oriented Programming

Chapter 7
Exceptions

Dr. Helei Cui

14 May 2024

Slides partially adapted from lecture notes by Cay Horstmann

Contents

7.1 Dealing with Errors

- When an error occurs, your program can:
  - Return to a safe state and allow the user to execute other commands.
  - Save the user's work and terminate the program.

- What kind of errors do you need to consider?
  1. User input errors.
  2. Device errors.
  3. Physical limitations.
  4. Code errors.

7.1 Dealing with Errors

- What can you do when an error occurs?
  1. Return an error code.
  2. Terminate the program.
  3. Throw an exception.

- Exceptions have their own syntax and are part of a special inheritance hierarchy.

7.1.1 The Classification of Exceptions

- **In Java, an exception object is always an instance of a class derived from `Throwable`.**
- You could create your own exception classes if those built into Java do not suit your needs.

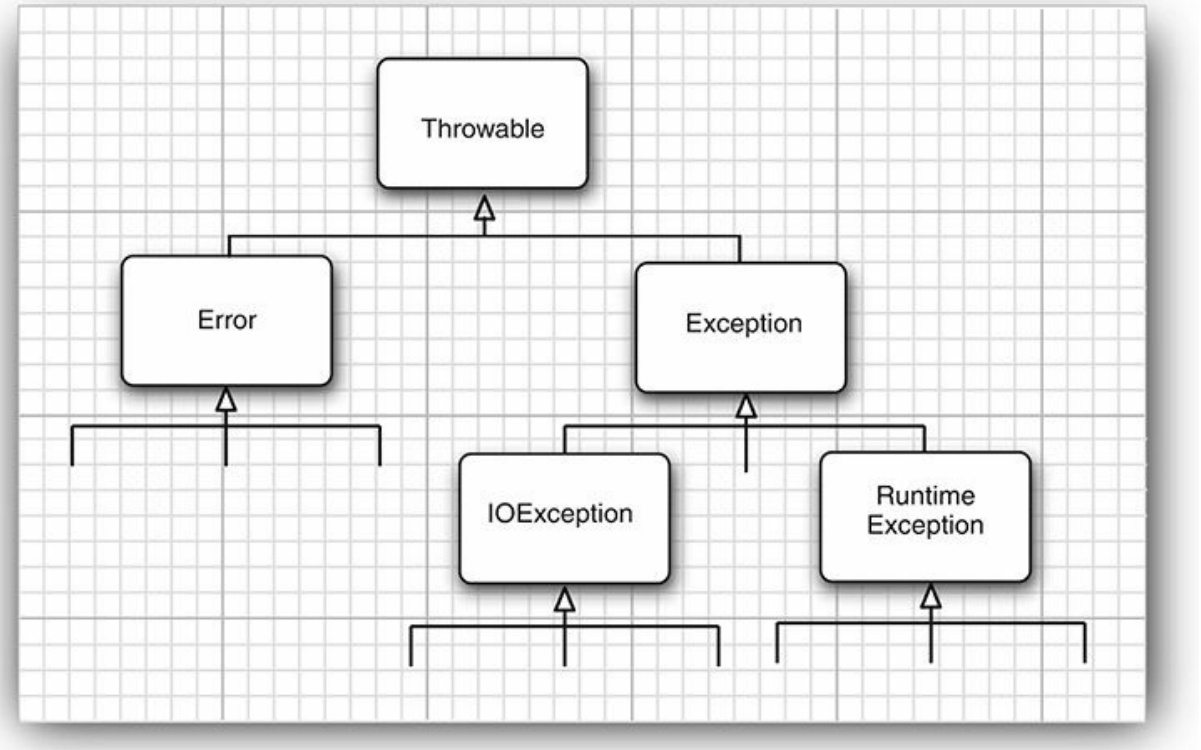

Figure 7.1 is a simplified diagram of the exception hierarchy in Java.

Error

- The `Error` hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system.
  - 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.
  - These situations are quite rare.

Exception

- **RuntimeException:** happens when you made a programming error.
  - A bad cast
  - An out-of-bounds array access
  - A null pointer access
- **Other exception:** occurs because a bad thing happened, e.g., an I/O error.
  - Trying to read past the end of a file
  - Trying to open a file that doesn't exist
  - Trying to find a Class object for a string that does not denote an existing class

The rule "If it is a RuntimeException, it was your fault" works pretty well.

Exception

The rule "If it is a RuntimeException, it was your fault" works pretty well.

- You could have avoided that `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.

- Any exception that derives from the class `Error` or the class `RuntimeException` is *unchecked* exception. All other exceptions are called *checked* exceptions.
  - The compiler checks that you provide exception handlers for all checked exceptions.

7.1.2 Declaring Checked Exceptions

- 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."
  - For example, 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 where your method can throw an exception is the header of the method.
  - For example, here is the declaration of one of the constructors of the `FileInputStream` class from the standard library.

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

7.1.2 Declaring Checked Exceptions

- There are four situations that an exception is thrown:
  1. Call a method that throws a checked exception.
  2. Detect an error and throw a checked exception with the `throw` statement.
  3. Make a programming error, such as `a[-1] = 0` that gives rise to an unchecked exception.
  4. An internal error occurs in the virtual machine or runtime library.
- If you write a method that might throw such an exception, you need to declare that fact.

7.1.2 Declaring Checked Exceptions

- Add a throws clause:

```
public Image loadImage(String s) throws IOException
```

- A throws clause can list multiple exceptions:

```
public Image loadImage(String s) throws FileNotFoundException,
EOFException
```

- Don't declare unchecked exceptions:

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

  - **Instead, fix your code so that this doesn't happen!**

7.1.2 Declaring Checked Exceptions

- In summary, a method must declare all the checked exceptions that it might throw.
  - Unchecked exceptions are either beyond your control (Error) or result from conditions that you should not have allowed in the first place (`RuntimeException`).
  - If your method fails to faithfully declare all checked exceptions, the compiler will issue an error message.
  - Of course, as you have already seen in quite a few examples, instead of declaring the exception, you can also catch it. Then the exception won't be thrown out of the method, and no throws specification is necessary.

When a method in a class declares that it throws an exception that is an instance of a particular class, it may throw an exception of that class or of its subclasses.

7.1.3 How to Throw an Exception

- Suppose something terrible happened in your code. You read a header that promised *Content-length: 1024*, but you got an end of file after 733 characters.
  - You may decide this situation is so abnormal that you want to throw an exception.
- Find an exception type to throw.
  - The Java library has an `EOFException` with description: **"Signals that an EOF has been reached unexpectedly during input."**
- Construct an object and throw it:

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

- Or, if you prefer:

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

7.1.3 How to Throw an Exception

- Here is how it all fits together:

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

- Or better, provide a reason:

```
String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);
```

7.1.3 How to Throw an Exception

- As you can see, throwing an exception is easy if one of the existing exception classes works for you. In this case:
  1. Find an appropriate exception class.
  2. Make an object of that class.
  3. 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.**

7.1.4 Creating Exception Classes

- Create your own exception class if your situation isn't covered by an exception in the standard library.
  - Just derive it from `Exception`, or from a child class of `Exception` such as `IOException`.

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

- Then you can throw an object of your own exception type:

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

Contents

7.2.1 Catching an Exception

- If an exception is thrown, and nobody catches it, the program will terminate and print a message to the console.
- Use a `try/catch` block to catch an exception:

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

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

7.2.1 Catching an Exception

- 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.

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

7.2.1 Catching an Exception

- Only do this if you can actually do something useful when the exception occurs.

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

- **There is no shame in propagating exceptions.**
- **One exception**: Sometimes you need to catch an exception when you override a method that is declared to throw no checked exceptions.
  - You are not allowed to add more throws specifiers to a subclass method than are present in the superclass method.

7.2.2 Catching Multiple Exceptions

- You can `catch` multiple exceptions in separate catch clauses:

```
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
}
```

- To find out more about the object

```
e.getMessage() // to get the detailed error message
e.getClass().getName() // to get the actual type of the exception object
```

- Work with the inheritance hierarchy of exceptions: `Catch` more specific exceptions before more general ones.

New Feature of Java 7

- As of Java 7, you can catch multiple exception types in the same `catch` clause.
  - For example, suppose that the action for missing files and unknown hosts is the same. Then you can combine the `catch` clauses:

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

  - This feature is only needed when catching exception types that are not subclasses of one another.

Notes

- When you catch multiple exceptions, the exception variable is implicitly `final`.
  - For example, you cannot assign a different value to `e` in the body of the clause.

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

- Catching multiple exceptions doesn't just make your code look simpler but also more efficient.
  - The generated bytecodes contain a single block for the shared `catch` clause.

7.2.3 Rethrowing and Chaining Exceptions

- Sometimes you want to `catch` an exception and rethrow it as a different type:

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

- **Better choice:** Set the original exception as the cause.

```
. . .
catch (SQLException original) {
    var e = new ServletException("database error");
    e.initCause(original);
    throw e;
}
```

- The cause can later be retrieved with the `getCause` method.

```
Throwable original = caughtException.getCause();
```

7.2.3 Rethrowing and Chaining Exceptions

- If you just want to log an exception and rethrow it without any change:

```
try {
    access the database
} catch (Exception e) {
    logger.log(level, message, e);
    throw e;
}
```

7.2.4 The `finally` Clause

- Suppose your code writes a resource that needs to be relinquished:

```
var in = new FileInputStream(. . .);
. . .
in.close();
```

- If the . . . code throws an exception, the in.close() statement is never executed.
- **Remedy:** Put it in a `finally` clause:

```
InputStream in = . . .;
try {
    . . .
} finally{
    in.close();
}
```

- You can use the `finally` clause without a `catch` clause.

7.2.4 The `finally` Clause

- Let's look at the three possible situations in which the program will execute the `finally` clause.

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

7.2.4 The `finally` Clause

- 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.

```
InputStream in = . . .;
try {
    try {
        code that might throw exceptions
    } finally{
        in.close();
    }
} catch (IOException e) {
    show error message
}
```

7.2.5 The try-with-Resources Statement

- As of Java 7, there is a useful shortcut to the code pattern.

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

- The Resource class must implement the AutoCloseable interface, which has a single method:

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

- The try-with-Resources statement has the form in its simplest variant:

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

7.2.5 The try-with-Resources Statement

- You can specify multiple resources.

```
try (var in = new Scanner (
        new FileInputStream("/usr/share/dict/words"), StandardCharsets.UTF_8);
        var out = new PrintWriter("out.txt", StandardCharsets.UTF_8)) {
    while (in.hasNext())
        out.println(in.next().toUpperCase());
}
```

- **No matter how the block exits, both `in` and `out` are closed.**
- As of Java 9, you can provide previously declared effectively final variables in the `try` header:

```
public static void printAll(String[] lines, PrintWriter out) {
    try (out) { // effectively final variable
        for (String line : lines)
            out.println(line);
    } // out.close() called here
}
```

7.2.5 The try-with-Resources Statement

- A difficulty arises when the `try` block throws an exception and the `close` method also throws an exception.
  - The `try`-with-resources statement handles this situation quite elegantly.
  - The original exception is rethrown, and any exceptions thrown by `close` methods are considered "suppressed."
  - They are automatically caught and added to the original exception with the `addSuppressed` method.
  - If you are interested in them, call the `getSuppressed` method which yields an array of the suppressed expressions from `close` methods.

You don't want to program this by hand. Use the try-with-resources statement whenever you need to close a resource.

7.2.6 Analyzing Stack Trace Elements

- When an exception terminates a program, a ***stack trace*** is displayed.
  - List of pending method calls.
- You can access the text description of a ***stack trace***:

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

- You can iterate over the stack frames with the `StackWalker` class:

```
StackWalker walker = StackWalker.getInstance();
walker.forEach(frame -> analyze frame)
```

- If you want to process the `Stream<StackWalker.StackFrame>` lazily, call

```
walker.walk(stream -> process stream)
```

Contents

Tips for Using Exceptions

1. Exception handling is not supposed to replace a simple test.

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

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

2. Do not micromanage exceptions.

```
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
}
```

Tips for Using Exceptions

3. **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`.
   - Respect the difference between checked and unchecked exceptions.
   - Do not hesitate to turn an exception into another exception that is more appropriate.
4. **Do not squelch exceptions:**

```
public Image loadImage(String s) {
    try {
        code that threatens to throw checked exceptions
    } catch (Exception e){

    } // so there
}
```

Tips for Using Exceptions

5. **When you detect an error, "tough love" works better than indulgence.**
   - When something is very wrong, throw an exception.
   - Don't return an error code or a dummy value.
   - Return values must be handled by the caller. Exceptions can be handled anywhere upstream.
6. **Propagating exceptions is not a sign of shame.**
   - Don't try to handle an exception that you can't remedy.
   - Just let it be rethrown so that it can reach a competent handler.

```
public void readStuff(String filename) throws IOException {
    var in = new FileInputStream(filename, StandardCharsets.UTF_8);
    . . .
}
```

These two rules can be summarized as: "throw early, catch late."

Recap

- 7.1 Dealing with Errors
- 7.2 Catching Exceptions
- 7.3 Tips for Using Exceptions