Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Chapter 7: Exceptions, Assertions, and Logging

7.1 Dealing with Errors

If an operation cannot be completed because of an error, the program ought to either

- **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.

7.1 Dealing with Errors

What sorts of problems do you need to consider?

- User input errors.
- Device errors.
- Physical limitations.
- Code errors.

Traditionally:

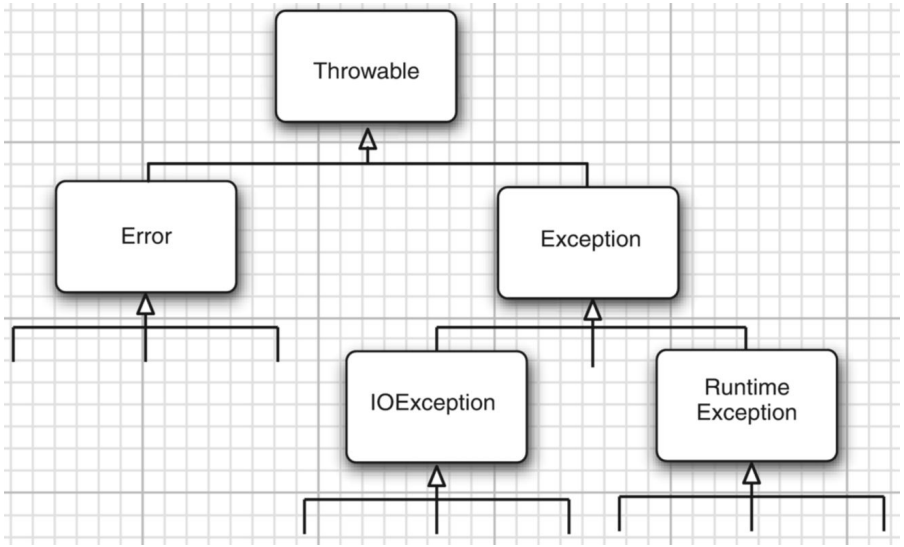
- return a special error code that the calling method analyzes.
- A method returning an integer cannot simply return -1 to denote the error since -1 might be a perfectly valid result.

=> Java throws an object that encapsulates the error information. The exception-handling mechanism search for **an exception handler**

7.1.1 The Classification of Exceptions

- **Error**: **internal** errors and **resource exhaustion** situations inside the Java runtime system => **should not throw an object of this type**
- **Exception**: exceptions that derive from RuntimeException and others.
- General rule:

A RuntimeException happens because you made a programming error.



7.1.1 The Classification of Exceptions

RuntimeException

- A bad cast
- An out-of-bounds array access
- A null pointer access

Others:

- 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

7.1.1 The Classification of Exceptions

- **Unchecked exception**: any exception that derives from the class **Error** or the class **RuntimeException**
- **Checked exception**: others

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**.
  - E.g., code that attempts to read from a file knows that the file might not exist or that it might be empty.

- We need to advertise the **throw** statement:
  - call a method that throws a checked exception, for example, the FileInputStream constructor.
  - detect an error and throw a checked exception.
  - …

```
public FileInputStream(String name) throws FileNotFoundException

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

7.1.4 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;
}
```

7.2 Catching Exceptions

7.2.1 Catching an Exception

- throw an exception: throw
- handle it: **try … catch**

*** If a method overrides a superclass method which throws no exceptions => catch each checked exception.

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

7.2.1 Catching an Exception

```
public void readVer1(String filename) throws IOException {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1) {
                // process input
        }
}

public void readVer2(String filename) {
        try {
                InputStream in = new FileInputStream(filename);
                int b;
                while ((b = in.read()) != -1) {
                        // process input
                }
        } catch (IOException exception) {
                exception.printStackTrace();
        }
}
```

7.2.2 Catching Multiple Exceptions

```
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
}
```

7.2.2 Catching Multiple Exceptions

The exception object may contain information about the nature of the exception.

- find out more about the object: e.getMessage()
- get the actual type of the exception object: e.getClass().getName()

7.2.2 Catching Multiple Exceptions

- Catch multiple exception types in the same catch clause: apply to catch exception types that are not subclasses of one another

```
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
}
```

7.2.3 Rethrowing and Chaining Exceptions

- throw an exception in a catch clause when wanting to change the exception type.

```
try {
    // access the database
     // statement 2;
     // statement 3;
}
catch (SQLException e) {
     // throw new ServletException("database error: " + e.getMessage());
}
…
```

- 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

- When your code throws an exception, it stops processing the remaining code and exits the method.
- If the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up.

=> to catch and rethrow all exceptions; **finally** clause

The **finally** clause executes whether or not an exception was caught.

7.2.4 The finally Clause

```
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
```

- Throws no exceptions: 1, 2, 5, and 6
- Throws an exception that is caught in a catch clause: 1, 3, 4, 5, and 6.
- Throws an exception that is not caught in any catch clause: 1, 5

7.2.4 The finally Clause

- Use the finally clause without a catch clause

```
InputStream in = . . .;
try
{
    //code that might throw exceptions,
    //need to catch in another clause
} finally {
    in.close();
}
```

7.2.4 The finally Clause

- try/catch and try/finally blocks

```
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

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

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

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

7.2.6 Analyzing Stack Trace Elements

A stack trace is a list of all pending method calls at a particular point in the execution of a program.

Full codes: v1ch07.stackTrace

7.3 Tips for Using Exceptions

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

```
if (!s.empty()) s.pop();
vs.
try {
        s.pop();
}catch (EmptyStackException e) {
}
```

7.3 Tips for Using Exceptions

- Do not micromanage exceptions: don't wrap every statement in a separate try block

```
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
}
```

7.3 Tips for Using Exceptions

- 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.
  - Respect the difference between checked and unchecked exceptions.

- Do not squelch exceptions.

```
public Image loadImage(String s) {
    try {
        // code that threatens to throw checked exceptions
    }catch (Exception e){ // silently ignore exception
        // …
    } // so there
}
```

7.3 Tips for Using Exceptions

- Propagating exceptions is not a sign of shame: higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.

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

7.4 Using Assertions

7.4.1 The Assertion Concept

```
double y = Math.sqrt(x);  // x > 0

if (x < 0) throw new IllegalArgumentException("x < 0");

// this code stays in the program, even after testing is complete
// run quite a bit slower if lots of checks like throw statements.
```

7.4.1 The Assertion Concept

The assertion mechanism allows to:

- check during testing
- automatically removed in the production code

Two forms: evaluate the condition and throw an AssertionError if it is false

```
assert condition;
```

and

```
assert condition: expression;
```

E.g,

```
assert x >= 0;
assert x >= 0 : x;
```

7.4.2 Assertion Enabling and Disabling

- By default, assertions are disabled, the class loader strips out the assertion code so that it won't slow execution.
- Enable by:

```
java -enableassertions MyApp
java -ea MyApp
```

7.4.3 Using Assertions for Parameter Checking

Three mechanisms to deal with system failures:

- Throwing an exception
- Logging
- Using assertions: only be used to locate internal program errors during testing.

7.4.4 Using Assertions for Documenting Assumptions

```
if(i%3==0)
    ...
else if (i % 3 == 1)
    ...
else // (i % 3 == 2)
    ...

if(i%3==0)
    ...
else if (i % 3 == 1)
    ...
else {
    assert i % 3 == 2;
... }
```

- Think through the issue thoroughly with **assertion** instead of **comments**.
- Self-test for the programmer.

7.5 Logging

Three mechanisms to deal with system failures:

- Throwing an exception
- Logging: a strategic tool for the entire lifecycle of a program
- Using assertions

7.5.1 Basic Logging

- For simple logging, use the global logger and call its info method:
```
Logger.getGlobal().info("File->Open menu item selected");
```

- By default: the output as
```
May 10, 2013 10:12:15 PM LoggingImageViewer file
Open INFO: File->Open menu item selected
```

- Call at an appropriate place (such as the beginning of main), all logging is suppressed.
```
Logger.getGlobal().setLevel(Level.OFF);
```

7.5.2 Advanced Logging

- Call the getLogger method to create or retrieve a logger:

```
private static final Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

- Seven logging levels: top 3 are default logged
  - SEVERE
  - WARNING
  - INFO
  - CONFIG: debugging
  - FINE: debugging
  - FINER: debugging
  - FINEST: debugging

- `myLogger.setLevel(Level.FINE);` //FINE and top 3 are logged.

- `Level.ALL` to turn on logging for all levels or `Level.OFF` to turn all logging off

7.5.2 Advanced Logging

- Log unexpected exceptions.

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)

if(...) {
     IOException exception = new IOException(". . .");
     logger.throwing("com.mycompany.mylib.Reader", "read", exception);
     throw exception;
}

try {
     ...
}catch (IOException e) {
     Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

7.5.3 Changing the Log Manager Configuration

The default configuration file is located at

```
jre/lib/logging.properties
```

Use another file, set the java.util.logging.config.file property to the file location by starting your application with

```
java -Djava.util.logging.config.file=configFile MainClass
```

Change the default logging level, edit the configuration file, and modify the line

```
.level=INFO
```

7.5.4 Localization

- Localize logging messages so that they are readable for international users.
- Locale-specific information is contained in **resource bundles**.
- Add mappings to a resource bundle, supply a file for each locale.
  - English mappings: com/mycompany/logmessages_**en**.properties
  - German mappings: com/mycompany/logmessages_**de**.properties.
  - **en** and **de**: language codes

`Code: v1ch07.logging.Localization`

7.5.5 Handlers

- By default, loggers send records to a **ConsoleHandler** that prints them to the **System.err** stream.
- Handlers also have a logging level.
- For a record to be logged, its logging level must be above the threshold of both the logger and the handler.

```
FileHandler handler = new FileHandler("log.txt");
handler.setLevel(Level.FINE);

logger.addHandler(handler);
```

7.5.6 Filters

- Records are filtered according to their logging levels.
- Each logger and handler can have an optional filter to perform additional filtering by implementing the interface Filter

```
public interface Filter {
    public boolean isLoggable(LogRecord record);
}
```

- Install a filter into a logger or handler, simply call the setFilter method.

```
logger.setFilter(newFilter);
```

- Analyze the log record, using any criteria that you desire, and return true for those records that should be included in the log.

7.5.7 Formatters

- ConsoleHandler and FileHandler classes emit the log records in text and XML formats, but we define our own formats.

```
SimpleFormatter formatter = new SimpleFormatter();
```

- Call the setFormatter method to install the formatter into the handler.

```
handler.setFormatter(formatter);
```