Error, Logging, Exception Handling

Errors

- Errors exist
  - Syntax errors
    - Easy to handle
  - Run-time Errors
    - At the least
      - Notify the user of an error;
      - Save all work;
      - Allow users to gracefully exit the program.

Java Error Management

- Exception Handling
- Assertions
- Logging

Exceptions

- 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;
  - or 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 save the user's work and exit cheerfully.
- The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation.

Errors

- User input errors: In addition to the inevitable typos, some users like to blaze their own trail instead of following directions.
  - Suppose, for example, that a user asks to connect to a URL that is syntactically wrong. Your code should check the syntax, but suppose it does not. Then 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. For example, a printer may run out of paper during printing.

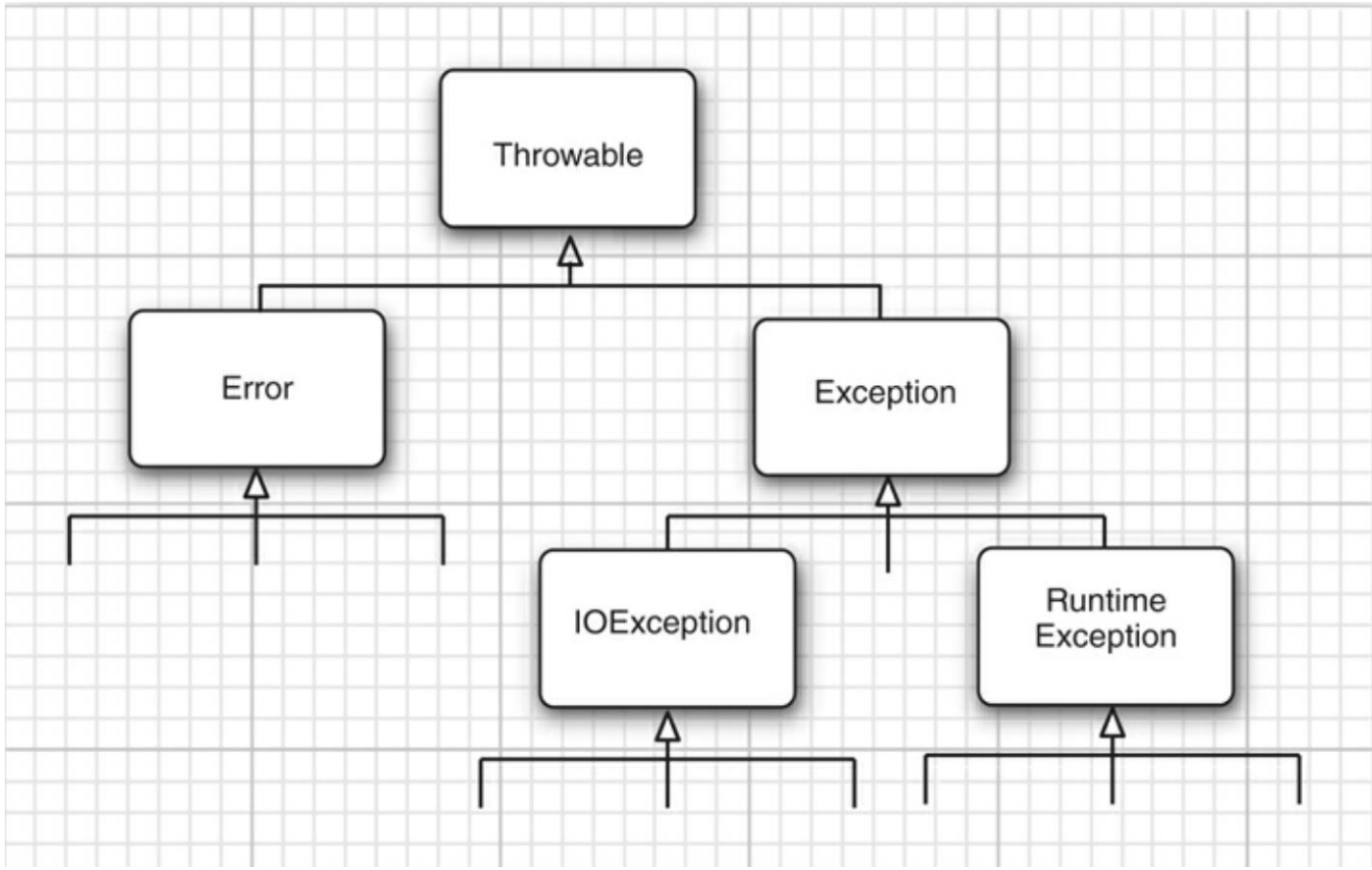
Errors

- Physical limitations: Disks can fill up; you can run out of available memory.
- Code errors: A method may not perform correctly. For example, 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
  - Trying to pop an empty stack

Java Exception Handling

- Any method that throws an exception is a potential death trap. If no handler catches the exception, the current thread of execution terminates.
- 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 does not return a value. Instead, it throws an object that encapsulates the error information.
- Note that the method exits immediately; it does not return its normal (or 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.

Classification of Exceptions



Types of Exceptions

Exceptions that inherit from RuntimeException

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

- Programmer is responsible

Exceptions that do not inherit from RuntimeException

- 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

Exceptions

Checked

- All other exceptions are called checked exceptions.
- The compiler checks that you provide exception handlers for all checked exceptions.

Unchecked

- The Java Language Specification calls any exception that derives from the class Error or the class RuntimeException an unchecked exception.

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 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

Unchecked Exceptions

- You **should not** advertise unchecked exceptions inheriting from RuntimeException.

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

- 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

Unchecked Exceptions

- 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.

How to throw an exception

- Find an appropriate exception class.
- Make an object of that class.
- Throw it.

```
while (. . .)
  {
    if (!in.hasNext()) // EOF encountered
    {
      if (n < len)
        throw new EOFException();
    }
    . . .
  }
```

Creating Exception Classes

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

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

Catching Exceptions

```
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 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.

Example

```
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();
  }
}
```

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

Caller of the read() to catch exception

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
}
```

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

Changing the type of Exception

The finally Clause

- When your code throws an exception, it stops processing the remaining code in your method and exits the method.
  - What if the method has acquired some local resource, which only this method knows about, and that resource must be cleaned up?
  - One solution is to catch all exceptions, carry out the cleanup, and rethrow the exceptions.
    - It is tedious because you need to clean up the resource allocation in two places–in the normal code and in the exception code.
- The finally clause can solve this problem.
- The code in the finally clause executes whether or not an exception was caught.

Example

```
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
```

- Finally can be used without a catch

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

Analyzing Stack Trace Elements

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

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

Or

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

Using Assertions

- The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.

```
double y = Math.sqrt(x);
```

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

Java assert

- assert condition
- assert condition : expression;
- E.g.
  - To assert that x is non-negative
    - assert x >= 0;
    - assert x >= 0 : x;

When should you choose assertions?

- Assertion failures are intended to be fatal, unrecoverable errors.
- Assertion checks are turned on only during development and testing.

Enabling assertions

- By default, assertions are disabled.
- Enable them by running the program with the -enableassertions or -ea option
- E.g.
  - java -enableassertions MyApp

Logging

- Cleaner alternative of “printf” statements
- Advantages
  - It is easy to suppress all log records or just those below a certain level, and just as easy to turn them back on.
  - Suppressed logs are very cheap, so there is only a minimal penalty for leaving the logging code in your application.
  - Log records can be formatted in different ways–for example, in plain text or XML.
  - Applications can use multiple loggers, with hierarchical names such as com.mycompany.myapp, similar to package names.

Basic Logging

- Logger.getGlobal().info("File->Open menu item selected");

- Output
  - May 10, 2013 10:12:15 PM LoggingImageViewer fileOpen
  - INFO: File->Open menu item selected

Logging levels

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

- Choose an appropriate level
- Logging could be verbose

Reference

- Interfaces in Java – GeeksforGeeks
- Horstmann, Cay S.. Core Java, Volume I (p. 661). Pearson Education. Kindle Edition.