西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Object Oriented Programming

Chapter 7
Exceptions

Dr. Muhammad Umar Farooq Qaisar

15th April 2025

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

1. **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.
2. **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.
3. **Physical limitations**. Disks can fill up; you can run out of available memory.
4. **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, or trying to pop an empty stack are all examples of a code error.

7.1 Dealing with Errors

- What can you do when an error occurs?
  1. Return an error code.
     - methods that read information back from files often return a -1 end-of-file value marker rather than a standard character. This can be an efficient method for dealing with many exceptional conditions. Another common return value to denote an error condition is the null reference.
     - Unfortunately, it is not always possible to return an error code. There may be no obvious way of distinguishing valid and invalid data. A method returning an integer cannot simply return `-1` to denote the error; the value `-1` might be a perfectly valid result.
  2. Terminate the program.
     - You may want to terminate the program, but this is not a good idea.

7.1 Dealing with Errors

1. Throw an exception.
   - 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.

- 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.

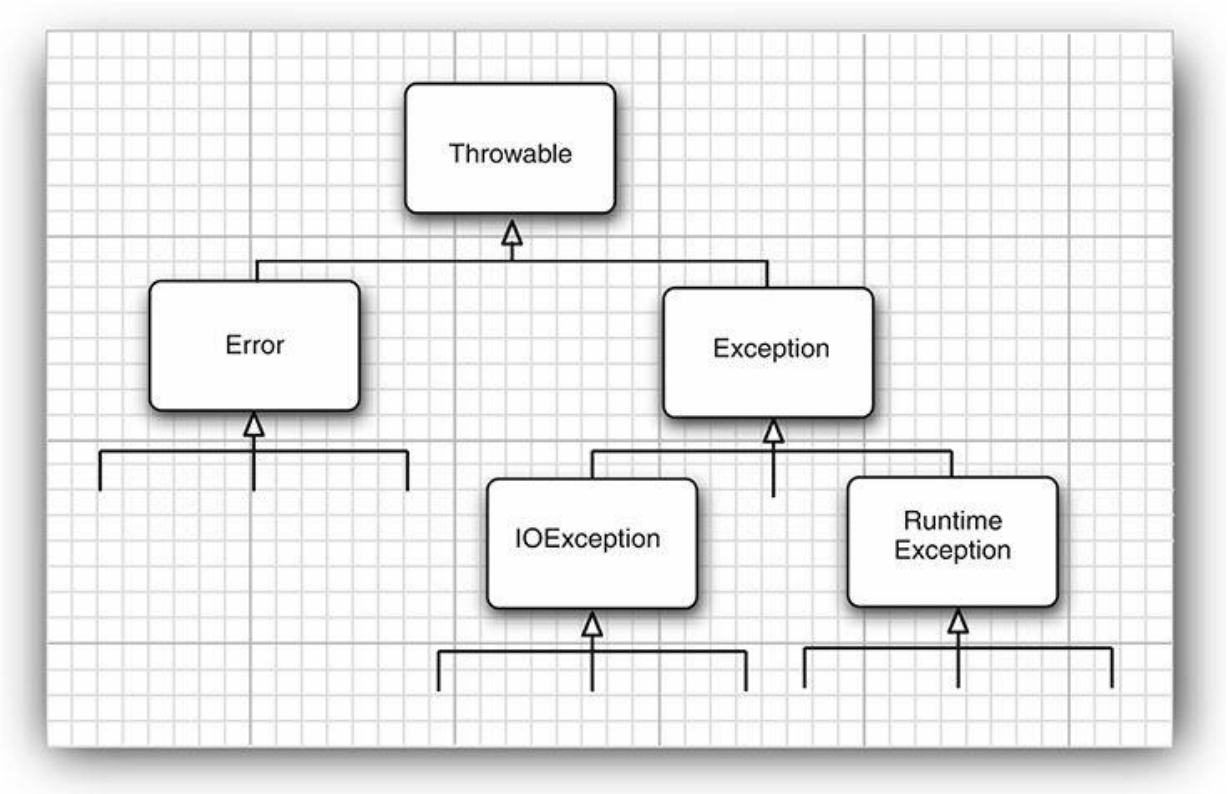

Figure 7.1 is a simplified diagram of the exception hierarchy in Java.

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.

```
int[] numbers = {10, 20, 30};
System.out.println(numbers[3]); // Throws ArrayIndexOutOfBoundsException (max index
is 2)
```

Fix (Check Bounds First):

```
int index = 3;
if (index >= 0 && index < numbers.length) { // Check if index is valid
    System.out.println(numbers[index]);
} else {
    System.out.println("Invalid index!");
}
```

Exception

- The `NullPointerException` would not have happened had you checked whether the variable was null before using it.

```
String name = null;
System.out.println(name.length()); // Throws NullPointerException (name is null)
```

Fix (Check for Null First):

```
if (name != null) { // Check if object exists
    System.out.println(name.length());
} else {
    System.out.println("Name is null!");
}
```

- 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.
     - Some methods declare that they might fail (e.g., reading a file).

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

  2. Detect an error and throw a checked exception with the `throw` statement.
     - You detect an error and forcefully throw an exception.

```
if (input < 0) {
    throw new IllegalArgumentException("Input must be positive!");}
```

  3. Make a programming error, such as `a[-1] = 0` that gives rise to an unchecked exception.
     - Bugs like out-of-bounds access (a[-1]) or NullPointerException.

7.1.2 Declaring Checked Exceptions

```
int[] a = new int[5];
a[-1] = 10; // Throws ArrayIndexOutOfBoundsException
```

4. An internal error occurs in the virtual machine or runtime library.
   - Rare, severe failures (e.g., OutOfMemoryError, StackOverflowError).

- 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.

7.1.2 Declaring Checked Exceptions

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.

- If a method says it can throw an exception of a certain type, Java also allows it to throw exceptions of any subclass of that type.

```
public void readFile() throws IOException {
    throw new FileNotFoundException("File not found");
}
```

- readFile() declares it throws IOException.
- FileNotFoundException is a subclass of IOException.
- So it's valid to throw FileNotFoundException.

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;
}
```

Top Hat Questions

Q1: Which is a checked exception in Java?

A) NullPointerException
B) ArrayIndexOutOfBoundsException
C) IOException
D) RuntimeException

Answer: C) IOException
Explanation: Checked exceptions (like IOException) must be declared or caught.

Top Hat Questions

Q2: What does the throws keyword do?

A) Catches an exception.
B) Declares that a method might throw an exception.
C) Ignores exceptions.
D) Terminates the program.

Answer: B) Declares that a method might throw an exception.
Explanation: throws delegate exception handling to the caller method.

Top Hat Questions

Q3: Which exception is thrown when accessing a null object's method?

A) ArrayIndexOutOfBoundsException
B) NullPointerException
C) ClassCastException
D) IllegalStateException

Answer: B) NullPointerException
Explanation: NullPointerException occurs when calling methods on null.

Top Hat Questions

Q4: How can you create a custom exception in Java?

A) Extend RuntimeException or Exception.
B) Use the new Exception() constructor.
C) Define it inside the try block.
D) Java doesn't support custom exceptions.

Answer: A) Extend RuntimeException or Exception.
Explanation: Custom exceptions must extend Exception (checked) or RuntimeException (unchecked).

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 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
}
```

7.2.2 Catching Multiple Exceptions

- 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), or e.getClass().getName() to get the actual type of the exception 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.
    - If one exception is a subclass of another, catching the parent exception will also handle the child.
    - The multi-catch feature (|) is only useful when the exceptions are unrelated (no inheritance between them).

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:
  - 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.
  - An example of such an exception type is the 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.
  - 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.

7.2.3 Rethrowing and Chaining Exceptions

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

```
. . .
catch (SQLException original) {                    // (1) Original error occurs
    var e = new ServletException("database error"); // (2) Create new wrapper exception
    e.initCause(original);                         // (3) Attach original to wrapper
    throw e;                                       // (4) Throw the wrapper
}
```

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

```
Throwable original = caughtException.getCause();  // (5) Extract the
original SQLException
```

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

7.2.3 Rethrowing and Chaining Exceptions

- **Original Error:**
  - An SQLException happens (e.g., database connection fails).
- **Wrap It:**
  - You create a new ServletException (to add context like "database error").
- **Chain Them:**
  - .initCause(original) links the SQLException to the new ServletException (like saying: "This ServletException happened because of that SQLException").
- **Throw the Wrapper:**
  - The ServletException (now carrying the SQLException inside it) propagates up.
- **Later, Unwrap It:**
  - When you catch the ServletException elsewhere, .getCause() retrieves the original SQLException you attached earlier.

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

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.
2. The code throws an exception that is caught in a catch clause—in our case, an IOException. For this, 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.
3. If the catch clause does not throw an exception, the program executes the first line after the finally clause. In this scenario, execution passes through points 1, 3, 4, 5, and 6.
4. 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.
5. 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.

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

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

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

Top Hat Questions

Q5: What is the primary purpose of a catch block in Java?

A) To prevent exceptions from occurring.
B) To handle and recover from specific exceptions.
C) To terminate the program gracefully.
D) To replace if-else statements.

Answer: B) To handle and recover from specific exceptions.
Explanation: catch blocks define how to recover from or log exceptions.

Top Hat Questions

Q6: What happens if an exception is thrown but not caught by any catch block?

A) The program ignores the exception and continues.
B) The JVM converts it to a warning.
C) The program crashes and prints a stack trace.
D) The exception is automatically rethrown.

Answer: C) The program crashes and prints a stack trace.
Explanation: Uncaught exceptions propagate up the call stack and terminate the program.

Top Hat Questions

Q7: Which of these is the correct syntax for catching multiple exceptions in one catch block?

A) catch (Exception1 || Exception2 e)
B) catch (Exception1, Exception2 e)
C) catch (Exception1 | Exception2 e)
D) catch (Exception1 & Exception2 e)

Answer: C) catch (Exception1 | Exception2 e)
Explanation: Java uses | to separate exception types in multi-catch (e.g., IOException | SQLException).

Top Hat Questions

Q8: When should you use a finally block?

A) Only when an exception occurs.
B) To execute code regardless of whether an exception occurs.
C) To replace catch blocks.
D) To throw new exceptions.

Answer: B) To execute code regardless of whether an exception occurs.
Explanation: finally runs whether the try succeeds or fails (e.g., closing resources).

Top Hat Questions

Q9: What is the output of this code?

```
try {
    throw new RuntimeException("Oops");
} catch (RuntimeException e) {
    System.out.println("Caught");
    throw e;
}
```

A) "Caught" (then the program ends normally).
B) "Caught" (then the program crashes with "Oops").
C) Compiler error (can't rethrow e).
D) No output.

throw new RuntimeException("Oops")

A RuntimeException is thrown explicitly.

catch (RuntimeException e)

The exception is caught, and "Caught" is printed.

throw e;

The same exception is rethrown, causing the program to terminate abnormally with the error.

Answer: B) To execute code regardless of whether an exception occurs.
Explanation: finally runs whether the try succeeds or fails (e.g., closing resources).

Top Hat Questions

Q10: What is the purpose of try-with-resources?

A) To catch multiple exceptions at once.
B) To automatically close resources (like files) after use.
C) To replace all catch blocks.
D) To hide exceptions.

Answer: B) To automatically close resources (like files) after use.
Explanation: try-with-resources ensures AutoCloseable resources (e.g., FileInputStream) are closed.