Chapter 15
Exception Handling

Chapter Goals
- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To learn how to catch exceptions
- To know when and where to catch an exception

Error Handling
- Traditional approach: Method returns error code
  - Problem: Forget to check for error code
    - Failure notification may go undetected
  - Problem: Calling method may not be able to do anything about failure
    - Program must fail too and let its caller worry about it
    - Many method calls would need to be checked

Continued...

Throwing Exceptions
- Exceptions:
  - Can't be overlooked
  - Sent directly to an exception handler—not just caller of failed method
- Throw an exception object to signal an exceptional condition
- Example: IllegalArgumentException:

  ```java
  illegal parameter value
  IllegalArgumentException exception
  throw new IllegalArgumentException("Amount exceeds balance");
  throw exception;
  ```

Continued...
Example

```java
public class BankAccount {
    public void withdraw(double amount) {
        if (amount > balance) {
            // Create an IllegalArgumentException
            IllegalArgumentException exception = new IllegalArgumentException("Amount exceeds balance");
            throw exception;
        }
        balance = balance - amount;
    }
}
```

Syntax 15.1: Throwing an Exception

```java
throw exceptionObject;
```

Example:
```java
throw new IllegalArgumentException();
```

Purpose:
To throw an exception and transfer control to a handler for this exception type.

Self Check

1. How should you modify the deposit method to ensure that the balance is never negative?
2. Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of `balance` afterwards?

Answers

1. Throw an exception if the amount being deposited is less than zero.
2. The balance is still zero because the last statement of the withdraw method was never executed.

Checked and Unchecked Exceptions

- Two types of exceptions:
  - Checked
    - The compiler checks that you don't ignore them
    - Due to external circumstances that the programmer cannot prevent
    - Majority occur when dealing with input and output
    - For example, `IOException`
Checked and Unchecked Exceptions

- Two types of exceptions:
  - Unchecked:
    - Extend the class `RuntimeException` or `Error`
    - They are the programmer's fault
  - Examples of runtime exceptions:
    - `NumberFormatException`
    - `IllegalArgumentException`
    - `NullPointerException`
  - Example of error: `OutOfMemoryError`

- Categories aren't perfect:
  - `Scanner.nextInt` throws `unchecked InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input
  - This choice makes the class easy to use for beginning programmers
  - Deal with checked exceptions principally when programming with files and streams

- For example, use a `Scanner` to read a file

```java
String filename = ...;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

But, `FileReader` constructor can throw a `FileNotFoundException`

- Two choices:
  - Handle the exception
  - Tell compiler that you want method to be terminated when the exception occurs
  - Use `throws` specifier so method can throw a checked exception

```java
public void read(String filename) throws IOException, ClassNotFoundException {
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    ...}
```

- For multiple exceptions:

```java
public void read(String filename)
    throws IOException, ClassNotFoundException
```

- Keep in mind inheritance hierarchy:
  - If method can throw an `IOException` and `FileNotFoundException`, only use `IOException`
- Better to declare exception than to handle it incompetently

**Syntax 15.2: Exception Specification**

`accessSpecifier returnType methodName[parameterType parameterName, . . .]
throws ExceptionClass, ExceptionClass, . . .`

Example:

```java
public void read(BufferedReader in) throws IOException
```

Purpose:
To indicate the checked exceptions that this method can throw
Self Check

3. Suppose a method calls the FileReader constructor and the read method of the FileReader class, which can throw an IOException. Which throws specification should you use?
4. Why is a NullPointerException not a checked exception?

Answer

3. The specification throws IOException is sufficient because FileNotFoundException is a subclass of IOException.
4. Because programmers should simply check for null pointers instead of trying to handle a NullPointerException.

Catching Exceptions

• Install an exception handler with try/catch statement
• try block contains statements that may cause an exception
• catch clause contains handler for an exception type

Example:

```java
try {
    String filename = ...;
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    String input = in.next();
    int value = Integer.parseInt(input);
    ... }
catch (IOException exception) {
    exception.printStackTrace();
}
catch (NumberFormatException exception) {
    System.out.println("Input was not a number");
}
```

Catching Exceptions

• Statements in try block are executed
• If no exceptions occur, catch clauses are skipped
• If exception of matching type occurs, execution jumps to catch clause
• If exception of another type occurs, it is thrown until it is caught by another try block

• catch (IOException exception) block
  • exception contains reference to the exception object that was thrown
  • catch clause can analyze object to find out more details
  • exception.printStackTrace(): printout of chain of method calls that lead to exception
Syntax 15.3: General Try Block

```java
try {
    statement
    statement
    . . .
} catch (ExceptionClass exceptionObject) {
    statement
    statement
    . . .
} catch (ExceptionClass exceptionObject) {
    statement
    statement
    . . .
} . . .
Continued...
```

Syntax 15.3: General Try Block

Example:

```java
try {
    System.out.println("How old are you?");
    int age = in.nextInt();
    System.out.println("Next year, you'll be " + (age + 1));
} catch (InputMismatchException exception) {
    exception.printStackTrace();
}
Purpose:
To execute one or more statements that may generate exceptions.
If an exception occurs and it matches one of the catch clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any catch clause, then skip the catch clauses.
```

Self Check

5. Suppose the file with the given file name exists and has no contents. Trace the flow of execution in the try block in this section.

6. Is there a difference between catching checked and unchecked exceptions?

Answers

5. The FileReader constructor succeeds, and in is constructed. Then the call in.nextInt() throws a NoSuchElementException, and the try block is aborted. None of the catch clauses match, so none are executed. If none of the enclosing method calls catch the exception, the program terminates.

Answers

6. No—you catch both exception types in the same way, as you can see from the code example on page 558. Recall that IOException is a checked exception and NumberFormatException is an unchecked exception.

The finally clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```java
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```
The finally clause

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed "no matter what"

Syntax 15.4: The finally clause

```java
try {
    Scanner in = new Scanner(reader);
    readData(in);
} finally {
    reader.close(); // if an exception occurs, finally clause
    // is also executed before exception is
    // passed to its handler
}
```

Continued...

Self Check

7. Why was the `reader` variable declared outside the `try` block?

8. Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.
Answers

7. If it had been declared inside the `try` block, its scope would only have extended to the end of the `try` block, and the catch clause could not have closed it.

8. The `FileReader` constructor throws an exception. The finally clause is executed. Since `reader` is null, the call to `close` is not executed. Next, a catch clause that matches the `FileNotFoundException` is located. If none exists, the program terminates.

Designing Your Own Execution Types

- You can design your own exception types—subclasses of `Exception` or `RuntimeException`

Designing Your Own Execution Types

- Make it an unchecked exception—programmer could have avoided it by calling `getBalance` first

Designing Your Own Execution Types

- If `amount > balance`
  
  ```java
  throw new InsufficientFundsException("withdrawal of " + amount + " exceeds balance of " + balance);
  ```

Self Check

9. What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

10. Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

Answers

9. To pass the exception message string to the `RuntimeException` superclass.

10. Exception or `IOException` are both good choices. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException`. 
A Complete Program

• Program
  • Asks user for name of file
  • File expected to contain data values
  • First line of file contains total number of values
  • Remaining lines contain the data
  • Typical input file:
    3
    1.45
    -2.1
    0.05

• What can go wrong?
  • File might not exist
  • File might have data in wrong format

• Who can detect the faults?
  • FileReader constructor will throw an exception when file does not exist
  • Methods that process input need to throw exception if they find error in data format

• What exceptions can be thrown?
  • FileNotFoundException can be thrown by FileReader constructor
  • IOException can be thrown by close method of FileReader
  • BadDataException, a custom checked exception class

• Who can remedy the faults that the exceptions report?
  • Only the main method of DataSetTester program interacts with user
    • Catches exceptions
    • Prints appropriate error messages
    • Gives user another chance to enter a correct file

---

File DataSetTester.java

```java
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class DataSetTester {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        DataSetReader reader = new DataSetReader();
        boolean done = false;
        while (!done) {
            try {
                System.out.println("Please enter the file name: ");
                String filename = in.next();
                double[] data = reader.readFile(filename);
                double sum = 0;
                for (double d : data) sum += d;
                System.out.println("The sum is " + sum);
                done = true;
            } catch (FileNotFoundException exception) {
                System.out.println("File not found.");
            } catch (BadDataException exception) {
                System.out.println("Bad data: " + exception.getMessage());
            }
        }
    }
}
```

---

Continued...
The \texttt{readFile} method of the \texttt{DataSetReader} class

- Constructs Scanner object
- Calls \texttt{readData} method
- Completely unconcerned with any exceptions

```java
public double[] readFile(String filename) throws IOException, BadDataException // FileNotFoundException is an IOException
{
    FileReader reader = new FileReader(filename);
    try
    {
        Scanner in = new Scanner(reader);
        readData(in);
    }
    finally
    {
        reader.close();
    }
    return data;
}
```

The \texttt{readFile} method of the \texttt{DataSetReader} class

- If there is a problem with input file, it simply passes the exception to caller

```java
private void readData(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
        throw new BadDataException("Length expected");
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];
    for (int i = 0; i < numberOfValues; i++)
        readValue(in, i);
    if (in.hasNext())
        throw new BadDataException("End of file expected");
}
```

The \texttt{readFile} method of the \texttt{DataSetReader} class

- Reads the number of values
- Constructs an array
- Calls \texttt{readValue} for each data value

```java
private void readValue(Scanner in) throws BadDataException
{
    if (!in.hasNextInt())
        throw new BadDataException("Length expected");
    int numberOValues = in.nextInt();
    data = new double[numberOValues];
    for (int i = 0; i < numberOValues; i++)
        readValue(in, i);
    if (in.hasNext())
        throw new BadDataException("End of file expected");
}
```

The \texttt{readFile} method of the \texttt{DataSetReader} class

- Checks for two potential errors
  - File might not start with an integer
  - File might have additional data after reading all values
- Makes no attempt to catch any exceptions
The `readFile` method of the `DataSetReader` class

```java
private void readValue(Scanner in, int i)
    throws BadDataException
{
    if (!in.hasNextDouble())
        throw new BadDataException("Data value expected");
    data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` calls `DataSetReader.readFile`
2. `readFile` calls `readData`
3. `readData` calls `readValue`
4. `readValue` doesn't find expected value and throws `BadDataException`
5. `readValue` has no handler for exception and terminates

Continued...

File `DataSetReader.java`

```java
/**
   * Reads a data set.
   * @param filename the name of the file holding the data
   * @return the data in the file
   */
public double[] readFile(String filename)
    throws IOException, BadDataException
{
    FileReader reader = new FileReader(filename);
    try
    {
        Scanner in = new Scanner(reader);
        readData(in);
    }
    finally
    {
        reader.close();
    }
return data;
```

Scenario

6. `readData` has no handler for exception and terminates
7. `readFile` has no handler for exception and terminates after executing finally clause
8. `DataSetTester.main` has handler for `BadDataException`; handler prints a message, and user is given another chance to enter file name

File `DataSetReader.java`

```java
/**
   * Reads all data.
   * @param in the scanner that scans the data
   */
private void readData(Scanner in)
    throws BadDataException
{
    if (!in.hasNextInt())
        throw new BadDataException("Length expected");
    int numberOfValues = in.nextInt();
    data = new double[numberOfValues];
    for (int i = 0; i < numberOfValues; i++)
        readValue(in, i);
```
Self Check

11. Why doesn’t the `DataSetReader.readFile` method catch any exceptions?

12. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

Answers

11. It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

12. `DataSetTester.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn’t catch it, so it propagates back to `main`, where it is caught.