Chapter 15: Failures and Exceptions

Objectives

- After studying this chapter you should understand the following:
  - The notion of program and method failure.
  - The Java exception mechanism: throwing and catching an exceptions, and exception propagation.
  - How to deal with method failure.

- Also, you should be able to:
  - catch exceptions in try-catch blocks;
  - define and use exceptions based on problem constraints.
Failure

- Inability of system to accomplish intended purpose.
- A method can fail for two reasons:
  - Logical error in its implementation.
  - Inability to obtain needed resource from environment.
- Programs containing error can do little about it at run-time, except offer helpful error message identifying and locating problem.

Failure

- A system may need resources from
  - hardware,
  - operating system,
  - file system,
  - network,
  - database, or
  - user to achieve its purpose.
- System may be unable to provide needed resource.
Failure

- **exception**: occurrence of a detectable, abnormal situation which may lead to system failure.
- Can design programs so that program itself will detect some logical errors.

The Java exception mechanism

- Allows for detecting, reporting, and handling exceptions.
- Not to be used to handle normal, expected events.
- Not just another control structure.
- Java detects certain run-time errors, such as:
  - attempts to divide by zero
  - use of a *null* reference when an object is required.
- System notifies program of error by *throwing an exception* from point at which error occurred.
The Java exception mechanism

- A thrown exception involves transfer of control:
  - processor stops executing current sequence of statements, and
  - begins executing statements at a different point in program.
- Exception is caught or handled at the point to which control is transferred.

The Java exception mechanism

```
processor executes statements

exception thrown

exception caught

program statements
```
The Java exception mechanism

- An exception is modeled as an instance of Java class *Throwable*.
- *Error* class and its subclasses represent conditions from which an ordinary program is not expected to recover.

The Java exception mechanism

- A few standard exceptions which are subclasses of *RuntimeException*:
  - *ArithmeticException*: an exceptional arithmetic situation has arisen, such as an integer division with zero divisor.
  - *ClassCastException*: attempt made to cast reference to an inappropriate type.
  - *IllegalArgumentException*: method invoked with invalid or inappropriate argument, or inappropriate object.
  - *NullPointerException*: attempt to use a null reference in case where an object reference was required.
  - *SecurityException*: a security violation was detected.
The Java exception mechanism

```
try {
    statements
    :
    :
} catch ( exception parameter1 ) {
    statements
    :
    :
} catch ( exception parameter2 ) {
    statements
    :
    :
} ...
```

Catching exceptions

- Syntactic code structure for the detection and handling of errors.
Catching exceptions

- Exceptions are objects

```java
try {
    statements
} catch ( ArithmeticException e) {
    statements
    :
} catch ( NullPointerException e ) {
    statements
    :
} catch ( Exception e ) {
    statements
    :
}
```

Catching exceptions

- Processor first performs statements of the `try` block.
  - If no exceptions occur, `try-catch` is complete, and `catch` ignores clauses.
  - If an exception is thrown during execution of `try` block, an attempt is made to match exception to `catch` clause parameters.
Catching exceptions

```java
try {
    i = i/i;
    j = 0;
    name = s.name();
    j = 1;
} catch ( ArithmeticException e ) {
    j = 3;
} catch ( NullPointerException e ) {
    j = 4;
} catch ( Exception e ) {
    if ( e instanceof IllegalArgumentException )
        j = 6;
    else
        j = 5;
} System.out.println( "The value of j is" + j );
```

Not-caught Exceptions: Propagated exceptions

- A not-caught exception:
  - Exception is thrown by execution of statement not part of a *try-catch*, or
  - Thrown exception does not match any *catch* clauses.
  - A not-caught exception is **propagated** up the “call chain”.
- If no method in the call chain catches exception, program terminates with error message.
Call-chain and not-caught exceptions

java.lang.ArithmeticException: / by zero
    at nim.Player.makeMove(Player.java:36)
    at nim.GameManager.play(GameManager.java:114)
    at nim.NimUI.start(NimUI.java:29)
    at nim.NimGame.main(NimGame.java:11)

Checked and unchecked exceptions

- The class `RuntimeException` and its subclasses are referred to as **unchecked** exception classes.
- Other exception classes are **checked** exception classes.
- A method’s specification must include a `throws` clause if it is possible for a **checked exception** to be thrown in method and method **does not** catch exception.
Checked and unchecked exceptions

- Take the specification:
  ```java
  public String readline() throws IOException
  ```
- This means that `IOException` is a checked exception. Method `readline` of `java.io.BufferedReader` can throw it.
- If `readLine` throws `IOException`, it is propagated to its caller, which must also include it in method specification if not caught, as it will be propagated to its own callers:
  ```java
  public void skip() throws IOException {
      String s;
      s = input.readLine();
  }
  ```

Using exceptions

- Server promises to fulfill a contract only if client satisfies preconditions.
  ```java
  public int indexOf (Object item)
  ```
  The index of the first occurrence of the specified item on this List, or -1 if this List does not contain the specified item.
- If we remove specification of returning a -1 if item not found in list, need to have precondition that item is in list.
- This puts an unreasonable burden on client.
Contract failures

- A method fails if it is unable to complete a contract even though its client has satisfied all preconditions.

- Three failure cases:
  - A logical error in the method.
  - The method is not able to obtain necessary resources.
  - The method invokes another method which fails.

Dealing with exceptions

- Two ways of dealing with failure of logical structures.
  - Clean up and report the failure to caller (by throwing an exception).
  - Attempt to correct situation that caused exception, and try again.
**FileReader example**

- `java.io.FileReader` constructor is specified as follows:
  ```java
  public FileReader (String fileName) throws
                     FileNotFoundException, SecurityException;
  ```

- Since `FileReader` may fail, and if client does not catch the exception thrown by `FileReader` constructor, it will be propagated to its caller.

- Thus `getSomeData` needs the following specification:
  ```java
  public void getSomeData () throws FileNotFoundException, SecurityException {
      FileReader in;
      in = new FileReader("DataFile");
      ...
  }
  ```

**FileReader example**

- `getSomeData` handles exceptions of `FileReader`, and re-throws them to its clients.
  ```java
  public void getSomeData () throws FileNotFoundException, SecurityException{
      FileReader in;
      try {
          in = new FileReader ("DataFile");
          ...
      } catch (FileNotFoundException e) {
          //cleanup
          throw e;    // throws it again to its caller
      }
      } catch (SecurityException e) {
          //cleanup
          throw e;    // throws it again to its caller
      }
  ```
Cleanup

- A method cannot know how its caller will respond to the exception.
- Caller might be able to recover.
- Important that method leave object in a **consistent** state (with all class invariants satisfied).
- Method should make sure that object is **consistent** before reporting failure to caller.

FileReader example (cont.)

- Suppose file is locked, but expect lock to be removed shortly.

```java
public void getSomeData () throws FileNotFoundException, SecurityException{
    FileReader in;
    boolean success = false; //Data file opened
    int tryNumber = 0;  // # of attempts to open datafile
    int delay = 5 * 1000; //wait in milli secs
    while (!success)
        try {
            tryNumber = tryNumber + 1;
            in = new FileReader("DataFile");
            success = true;
        } catch (SecurityException e) {
            if (tryNumber == 1)
                thisThread.sleep(delay);
            else
                throw e;
        }
}
```
Application defined exceptions

- Useful to define own exception classes to pass certain information to client.
- Example: throw exception if fail to get data.

```java
public class NoDataException extends Exception {
    public NoDataException () {
        super();
    }
    public NoDataException (String s) {
        super(s);
    }
}
```

```java
public void getData () throws NoDataException {
    try {
        in = new FileReader("DataFile");
        ...
    } catch (FileNotFoundException e) {
        //cleanup
        throw new NoDataException ("File does not exist");
    } catch (SecurityException e) {
        //cleanup
        throw new NoDataException ("File cannot be accessed");
    }
}
```
Application defined exceptions

- Exception class can pass state information from method detecting exception to method that handles it.

```java
public class BadDataException extends Exception {
    private int lineNumber; // stores line number where fault occurs

    public BadDataException (int lineNumber) {
        super();
        this.lineNumber = lineNumber;
    }

    public int lineNumber() {
        return this.lineNumber;
    }
}
```

Exceptions

- Structured as immutable objects: their interface includes no state-changing commands.
Dealing with logical errors

- Sometimes a logical error causes a method to produce reasonable but incorrect results.
- We can check preconditions, postconditions, and invariants.
- If a client invokes a method without preconditions being satisfied, it is an error.

Dealing with logical errors

- Assert statement used to verify preconditions.
- Two forms:
  
  ```java
  assert booleanExpression ;
  assert booleanExpression : expression ;
  ```

  ```java
  //Interchange list.get(i) and list.get(j)
  // require 0 <= i, j < list.size() ...
  private <Element> void interchange (List<Element> list, int i, int j) {
      assert 0 <= i && i < list.size();
      "precondition: illegal i";
      assert 0 <= j && j < list.size();
      "precondition: illegal j";
  ...
  ```
Dealing with logical errors

- We can use `assert` to check post-conditions, and invariants.
- Postconditions and invariants often too complex to verify with simple conditions.
- Postconditions
  - can be tricky to handle;
  - often they involve comparing an object’s state after method execution to the object’s state prior to execution.
- Including such checks depends on where we are in the development process.

Summary

- Addressed program failure.
- A method can fail for two fundamental reasons:
  - a logical error in its implementation (a programming “bug”); or
  - its inability to obtain some needed resource from the environment.
- Examined exception mechanism provided by Java to deal with failures.
Summary

- An exception mechanism is provided by the language for detecting, reporting, and handling failure.
- An exception is a detectable, abnormal situation which may lead to system failure, modeled by an instance of the Java class Exception.
- An Exception instance carries information about the exception from the point at which the exception occurred (is thrown) to the point at which it is handled (is caught).

Summary

- The language structure for handling exceptions is the try-catch statement.
- Exceptions thrown in the statements that comprise the try block can be handled in one of the catch clauses.
- An exception thrown in a method and not caught in the method is propagated to the method’s caller.
- A method fails if it cannot satisfy its contract even though the client has satisfied the method’s preconditions. A method that fails must not simply return to its client. It must inform the client of the failure by throwing an exception.
Summary

- A method fails if it cannot satisfy its contract even though the client has satisfied the method’s preconditions.
- A method that fails must not simply return to its client. It must inform the client of the failure by throwing an exception.

Summary

- When a client is notified of a server’s failure (by server’s throwing an exception), there are only two possible courses of action client can take.
  - attempt to correct the situation that caused the failure, and try again; or
  - report failure to its caller, by throwing or propagating an exception.
- Most often, the second alternative is the only one practical.
Summary

- An application can define its own exception classes, by extending the class `Exception` of one of its subclasses.
- Program defined exceptions can be useful in providing more specific information about the cause of the failure.

Summary

- Logical errors, by their very nature, can be difficult to detect.
- Can be useful, particularly during program development, to verify explicitly conditions such as preconditions that must hold in a correct program.