Exception Handling

Chapter 11

Logical Object Layering

A program viewed as a stack of objects (Rumbaugh’s layers and partitions)

- “Knowledge” of program is high at top, low at bottom
  - Method calls between objects on the same level (association)
  - Method calls to lower objects (dependency, aggregation, composition)
  - Method calls to higher objects rare (upcasting, polymorphism, interface, or static)
- Errors in lower classes are often passed up to higher classes for an appropriate response

Error Detection and Notification

Signaling errors to a higher calling scope

- Errors often cannot be dealt with at the point at which they are detected
  - “Library” routines (i.e., low objects) pass error information back up the calling stack, where appropriate actions are determined

```java
public void push(int data) {
    if (sp < StackSize)
        StackMem[sp++] = data;
    else
        ERROR // now what?
}

public int pop() {
    if (sp > 0)
        return StackMem[--sp];
    else
        ERROR // now what?
}
```

Exception

Three meanings determined from context

- An event that occurs during the execution of a program and disrupts the normal flow of control (i.e., an operation fails)
  - Caused by anomalous situations
  - Notifies the system of the situation
  - Cannot be ignored
- The name of a class that extends Throwable
- An object instantiated from Throwable or a subclass
  - Object notifies the system of and conveys or encapsulates information about an anomalous situation
  - Can have attributes or data
  - Can have unique methods; does have inherited methods
  - Dealing with these objects is called exception handling

Error-Type Hierarchy

The family of things that can be thrown

- Blue classes are “checked” and must be “claimed”
- Red classes are not “checked” and need not be “claimed”

Exception Categories

Highlights

- Error (and subclasses)
  - Internal system errors and resource exhaustion
  - Rare
  - Not checked (need not be claimed – also applied to subclasses)
  - Programmers should not throw objects of this type
- RuntimeException (and subclasses)
  - Programming errors (bugs?)
    - Bad cast
    - Out-of-bounds array access
    - Null pointer access
  - Not checked (need not be claimed – fix? – applied to subclasses)
- The rest of the Exception hierarchy
  - Caused by external circumstances
  - Are checked (must be claimed)
**Exception Summary**

When to claim an exception with **throws**

<table>
<thead>
<tr>
<th>Situation</th>
<th>Base Class</th>
<th>Claimed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java internal error</td>
<td>Error</td>
<td>Not Checked</td>
</tr>
<tr>
<td>Programming error</td>
<td>RuntimeException</td>
<td>Not Checked</td>
</tr>
<tr>
<td>Call a method that throws an exception</td>
<td>Exception</td>
<td>Checked</td>
</tr>
<tr>
<td>Detect an error and throw an exception</td>
<td>Exception</td>
<td>Checked</td>
</tr>
</tbody>
</table>

**Inherited Methods**

Defined in class **Throwable** (see Except.java)

- **public String getMessage()**
  <Short message about the error
  </by zero
- **public String toString()**
  <More detailed message about the error and the thrown object
  </java.lang.ArithmeticException: / by zero
- **public void printStackTrace()**
  <Detailed message and method call stack trace
  </java.lang.ArithmeticException: / by zero
  at Except.bad(Except.java:16)
  at Except.main(Except.java:10)

**Exception Handling Steps**

Java’s™ keywords

- **Claiming an exception**
  <**throws** Introduces a comma-separated list of exceptions that a method might throw
- **Throwing an exception**
  <**throw** Launch an exception object to indicate that an exception situation has occurred
- **Catching an exception**
  <**try** Define a block of code that anticipates exceptions
  <**catch** Define a block of code that deals with a specific exception
  <**finally** Define a block of code that executes regardless of whether an exception is generated or not

**User-Defined Exceptions**

Customized exceptions

- Use Java’s™ exception classes when possible
- May have instance variables, methods, and constructors
- Are derived from class **Exception**

  ```java
  public class Overflow extends Exception {
  private String message;
  public Overflow(String message) {
    this.message = message;
  }
  public void Display() {
    System.out.println(message);
  }
  }
  ```

**throws and Exception Lists**

Claiming or advertising errors with an exception specification

- **throws** introduces a comma-separated list of possible exceptions that may be generated within a method
- Checked exceptions must be advertised or claimed
- Unchecked exceptions are not advertised or claimed

  ```java
  public void push(int data) throws Overflow {
  ...
  }
  public int pop() throws Underflow {
  ...
  }
  ```

**throwing Exceptions**

Announcing errors

- An exception object is instantiated, constructed, and launched or thrown (manipulated by a thread)
- Execution resumes somewhere but not following the throw statement

  ```java
  int pop() throws Underflow {
  if (sp > 0)
    return StackMem[--sp];
  else
    throw new Underflow("EMPTY Stack");
  }
  ```
**try and catch Blocks**

Dealing with errors

- **try** block encloses code that may generate an exception
- **catch** block(s) deal with specific exceptions
  - Must catch all checked exceptions that are not claimed with throws
  - Can catch all exceptions that can be thrown
- When an exception is caught
  - Warn the user, deal with the exception, return the program to a known or stable state where execution can continue
  - Warn the user, clean up, perform an orderly shut-down

```
try
    { S.push(i);
      S.pop();
    }
catch (Overflow E)
    { System.err.print("ERROR: ");
      E.Display();
    }
catch (Underflow E)
    { System.err.print("ERROR: ");
      E.Display();
    }
```

**finally Block**

Cleaning up resources (See Except.java)

- Identifies a block that is always executed
- Follows the last catch block

```
try
    { x = 100 / x;
      if (x <= 1)
        { System.out.println("early return");
          return 1;
        }
    }
catch (ArithmeticException E)
    { System.out.println("Exception caught: " + E.getMessage());
    }
finally
    { System.out.println("Finally executed");
    }
System.out.println("normal return");
return x;
```

**finally and Execution Sequences**

Execution order for three cases

- No exception
  - All code in the try block
  - Code in finally block
  - Code following try/catch/finally
- Exception thrown and caught
  - Code in the try block before exception
  - Code in the catch block
  - Code in the finally block
  - Code following try/catch/finally (assuming no other errors)
- Exception thrown but not caught
  - Code in the try block before exception
  - Code in the finally block
  - Exception is thrown to a higher calling scope

**Possible finally Evaluation**

Orderly clean up

```
x == 0
catch
try
    x == 10
finally	normal return
x == 100
early return
x == 100
return 1
```

**Program Flow with Exceptions**

Tracing execution flow when exception occur

```
public static void function(myStack S) throws Overflow, Underflow
    { for (int i = 0; i <= 100; i++)  // one too many
        S.push(i);
    }
```

```
public static void main(String args[])
    { myStack S = new myStack( );
      try {
        function(S);
      }
      catch(Overflow E) {
      }
    }
```

**Program Flow Illustrated**

Where does execution resume?
Derived Exceptions

Catching a base class catches all of its children

- A method that claims a base class may throw a base or a derived exception object
- A catch block will catch base and derived exception objects

```
class Base extends Exception {...}
class Child extends Base {...}
try {
    // code that might throw an exception
} catch (Base E) // catches Base or Child exceptions
```

Rethrowing Exceptions

“Passing the buck”

- Perform local “clean up”

```
Graphics g = image.getGraphics();
try {
    // code that might throw an exception
} catch (MalformedURLException e) {
    g.dispose();
    throw e;
}
```

Termination vs Resumption

To repeat or not to repeat

- Exception handling theory has two basic models
  Termination (the Java™ and C++ model)
  Resumption (exception handler takes remedial action & reexecutes)

```
while (!done)
    try {
        /* might throw an exception */
        done = true;
    } catch (Bad B) {
        /* remedial action */
    } catch (Fatal F) {
        /* clean up */
        System.exit(1);
    }
```

try Blocks, Scope, and Initialization

Problems

```
try {
    int i = Integer.parseInt("100");
} catch(...) {
    // use i
}
```

Solutions

```
int i = 0;
try {
    i = Integer.parseInt("100");
} catch(...) {
    // use i
}
```

Miscellaneous Exception Concepts

Some suggestions

- Use exceptions to make a class as flexible as possible (i.e., it’s hard to know how a user will utilize your class)
- If the problem can be dealt with at the point of detection, use an if-else statement and handle it locally
- runtime exceptions do not appear in exception lists- fix them
  - Arithmetic
  - Null pointer
  - Array index out of range
- Exceptions are expensive and even though try/catch blocks resemble switches, they should not be used for flow control
- Don’t embed each statement in its own try block