Program Correctness and Efficiency

Following Koffmann and Wolfgang Chapter 2
Outline

• Categories of program errors
• Why you should catch exceptions
• The `Exception` hierarchy
  • Checked and unchecked exceptions
• The `try-catch-finally` sequence
• Throwing an exception:
  • What it means
  • How to do it
Outline (continued)

• A variety of testing strategies
• How to write testing methods
• Debugging techniques and debugger programs
• Program verification: assertions and loop invariants
• Big-O notation
  • What it is
  • How to use it to analyze an algorithm’s efficiency
Program Defects and “Bugs”

- An efficient program is *worthless* if it breaks or produces a wrong answer
- Defects often appear in software after it is delivered
- Testing cannot prove the absence of defects
- It can be difficult to test a software product completely in the environment in which it is used
- *Debugging*: removing defects
Major Categories of Defects

- Syntax and other in-advance errors
- Run-time errors and exceptions
- Logic Errors
Syntax Errors

• **Syntax errors**: grammatical mistakes in a program
• The *compiler detects* syntax errors
  • You *must* correct them to compile successfully
• Some common syntax errors include:
  • Omitting or misplacing braces, parentheses, etc.
  • Misplaced end-of-comment
  • Typographical errors (in names, etc.)
  • Misplaced keywords
Semantic Errors

- **Semantic errors**: may obey grammar, but violate other rules of the language
- The *compiler detects* semantic errors
  - You *must* correct them to compile successfully
- Some common semantic errors include:
  - Performing an incorrect operation on a primitive type value
  - Invoking an instance method not defined
  - Not declaring a variable before using it
  - Providing multiple declarations of a variable
  - Failure to provide an exception handler
  - Failure to *import* a library routine
Run-time Errors or Exceptions

- **Run-time errors**
  - Occur during program execution (run-time!)
  - Occur when the JVM detects an operation that it knows to be incorrect
  - Cause the JVM to throw an exception
- **Examples of run-time errors include**
  - Division by zero
  - Array index out of bounds
  - Number format error
  - Null pointer exceptions
Run-time Errors or Exceptions (continued)

<table>
<thead>
<tr>
<th>Run-time Exception</th>
<th>Cause/Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>Integer division by zero.</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>An attempt to access an element in an array with an index value (subscript) less than zero or greater than or equal to the array’s length.</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>An attempt to call a method with an argument of incorrect type or inappropriate format.</td>
</tr>
<tr>
<td>NumberFormatException</td>
<td>An attempt to convert a string that is not numeric to a number (real or integer).</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>An attempt to use a null reference value to access an object.</td>
</tr>
<tr>
<td>NoSuchElementException</td>
<td>An attempt to get a next token after all tokens were extracted from the string that was tokenized.</td>
</tr>
</tbody>
</table>
Logic Errors

• A *logic error* is programmer mistake in
  • the *design* of a class or method, or
  • the *implementation* of an algorithm

• Most logic errors
  • Are not syntax or semantic errors: get by the compiler
  • Do not cause run-time errors
  • Thus they are *difficult to find*

• Sometimes found through testing
• Sometimes found by users
Avoiding Logic Errors

- Work from a *precise* specification
- Strive for clarity and simplicity
- Consider “corner” / extreme cases
- Have reviews / walk-throughs: *other eyes*
- Use library/published algorithms where possible
- Think through pre/post conditions, invariants
- Be organized and careful in general
The Exception Class Hierarchy

• When an exception occurs, the *first* thing that happens is a `new` of a Java exception object
• Different exception classes have different rules
• **Throwable** is the *root superclass* of the exception class hierarchy
  • **Error** is a subclass of **Throwable**
  • **Exception** is a subclass of **Throwable**
    • **RuntimeException** is a subclass of **Exception**
The Class Throwable

- `Throwable` is the superclass of all exceptions
- All exception classes inherit its methods
### The Class Throwable (continued)

#### Table 2.2
Summary of Commonly Used Methods from the java.lang.Throwable Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String getMessage()</code></td>
<td>Return the detail message.</td>
</tr>
<tr>
<td><code>void printStackTrace()</code></td>
<td>Print the stack trace to System.err.</td>
</tr>
<tr>
<td><code>String toString()</code></td>
<td>Return the name of the exception followed by the detail message.</td>
</tr>
</tbody>
</table>
The Exception Class Hierarchy (2)

`Throwable` is the superclass of all exception classes
• **Error** is for things a program *should not* catch
  • Example: `OutOfMemoryError`
• **Exception** is for things a program *might* catch
  • **`RuntimeException`** is for things the VM might throw
    • It can happen anywhere: e.g., any object access can throw `NullPointerException`
    • So *not required* to catch it
  • **All others** must be either:
    • Explicitly *caught* or
    • Explicitly *mentioned as thrown* by the method
Exception Hierarchy Summary

- **Error**: don’t catch, unchecked
- **Exception**:
  - **RuntimeException**:
    - (Usually) don’t catch, unchecked
  - **All others**: checked, so must
    - Catch, or
    - Mention they may be thrown
Checked and Unchecked Exceptions

• Checked exceptions
  • Normally not due to programmer error
  • Generally beyond the control of the programmer
  • Examples: IOException, FileNotFoundException

• Unchecked exception may result from
  • Programmer error
  • Serious external condition that is unrecoverable
Checked and Unchecked Exceptions (2)

Figure 2.2
Exception Hierarchy Showing Selected Checked and Unchecked Exceptions

Unchecked
Some Common Unchecked Exceptions

- `ArithmeticException`
  - Division by zero, etc.
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`
  - Converting a “bad” string to a number
- `NullPointerException`
- `NoSuchElementException`
  - No more tokens available
Catching and Handling Exceptions

• When an exception is thrown, the normal sequence of execution is interrupted

• Default behavior, i.e., no handler
  • Program stops
  • JVM displays an error message

• The programmer may provide a handler
  • Enclose statements in a `try` block
  • Process the exception in a `catch` block
Example Handler

```java
InputStream in = null;
try {
    in = new FileInputStream(args[0]);
    ...
} catch (FileNotFoundException e) {
    System.out.printf("File not found: %s\n", name);
} catch (Throwable e) {
    System.err.println("Exception!");
    e.printStackTrace(System.err);
} finally {
    if (in != null) in.close();
}
```
Uncaught Exceptions

• Uncaught exception exits VM with a stack trace
• The stack trace shows
  • The sequence of method calls
  • Starts with throwing method
  • Ends at main

![Figure 2.3](image-url)

Example of a Stack Trace for an Uncaught Exception
The \texttt{try-catch} Sequence

\begin{itemize}
\item Avoiding uncaught exceptions
  \begin{itemize}
  \item Write a \texttt{try-catch} to handle the exception
  \item \textbf{Point}: prevent ugly program termination!
    \begin{itemize}
    \item Unpleasant for user
    \item Worse, may leave things messed up / “broken”
  \end{itemize}
  \end{itemize}
\item \texttt{catch} block is skipped if no exception thrown within the \texttt{try} block
\end{itemize}
Handling Exceptions to *Recover* from Errors

- Exceptions provide the opportunity to
  - *Report* errors
  - *Recover* from errors
- *User* errors common, and *should be recoverable*
- Most closely enclosing handler that *matches* is the one that executes
  - A handler matches if its class includes what’s thrown
- Compiler displays an error message if it encounters an unreachable *catch* clause
The **finally** block

- On exception, a **try** is abandoned
- Sometimes more actions **must** be taken
  - Example: Close an output file
- Code in a **finally** block is **always** executed
  - After the **try** finishes normally, or
  - After a **catch** clause completes
- **finally** is **optional**
Example of finally block

try {
    InputStream ins = ...;
    ... ins.read(); ...
}

} catch (EOFException e) {
    System.err.println("Unexpected EOF");
e.printStackTrace();
    System.exit(17);

} finally {
    if (ins != null) ins.close();
}
Throwing Exceptions

- Lower-level method can **pass exception through**
  - Can be caught and handled by a higher-level method
  - *Mark* lower-level method
    - Say it may throw a checked exception
    - Mark by **throws** clause in the header
  - May throw the exception in the lower-level method
    - Use a **throw** statement
- Particularly useful if calling module already has a handler for this exception type
Throwing Exceptions (2)

• Use a **throw** statement when you detect an error
• Further execution stops immediately:
  • Goes to closest suitable handler
  • May be a number of level of calls earlier
  • Does execute any **finally** blocks in the middle
Example of Throwing an Exception

/** adds a new entry or changes an old one
 * @param name the name to create/update
 * @param number the (new) number
 * @return the previous number, a String
 * @throws IllegalArgumentException if the number
 * is not in phone number format
 */

public String addOrChangeEntry(
    String name, String number) {
    if (!isPhoneNumberFormat(number)) {
        throw new IllegalArgumentException(
            "Invalid phone number: " + number);
    }
    ...
}
Another Example of Throwing an Exception

```java
public void accessLocalFile (String askingUser) throws CertificateException {
    ...
    if (user’s secure socket certificate bad) {
        throw new CertificateException(reason);
    }
    ...
}
```
Programming Style

• You can always avoid handling exceptions:
  • Declare that they are thrown, or
  • Throw them and let them be handled farther back

• But: usually best to handle instead of passing

• Guidelines:
  1. If recoverable here, handle here
  2. If checked exception likely to be caught higher up
     Declare that it can occur using a throws clause
  3. Don’t use throws with unchecked exceptions
     Use an @throws javadoc comment when helpful
Programming Style (2)

Don’t do this!

```java
try {...} catch (Throwable e) { }
```

- Omits arbitrary patches of code
  Can leave things in “broken” state
- No warning to user
- Leads to hidden, difficult to detect, defects
Handling Exceptions in Phone Dir Example

In `loadData`:
- `FileNotFoundException` from `FileReader` constructor
- `IOException` from `readLine`

In `PDConsoleUI`:
- `InputMismatchException` from `nextInt`

In `addOrChangeEntry`:
- `IllegalArgumentException` for empty `String`
Testing Programs

• A program with
  • No syntax/semantic errors, and
  • No run-time errors,
  • May still contain logic errors
• “Best” case is logic error that always executes
  • Otherwise, hard to find!
• Worst case is logic error in code rarely run

Goal of testing: Test every part of the code, on “good” and “bad”/”hard” cases
Structured Walkthroughs

- Most logic errors:
  - Come from the design phase
  - Result from an incorrect algorithm
- Logic errors sometimes come from typos that do not cause syntax, semantic, or run-time errors
  - Famous FORTRAN: DO 10 I = 1.100
  - Common C: if (i = 3) ...
- One way to test: hand-trace algorithm before implementing!
- Thus: Structured Walkthroughs
Structured Walkthroughs (2)

The Designer:
- Explains the algorithm to other team members
- Simulate its execution with them looking on

The Team:
- Verifies that it works
- Verifies that it handles all cases

Walkthroughs are helpful, but do not replace testing!
Testing Defined

• **Testing**:  
  - Exercising a program under controlled conditions  
  - Verifying the results  
• **Purpose**: detect program defects after  
  - All syntax/semantic errors removed  
  - Program compiles  
• No amount of testing can *guarantee* the absence of defects in sufficiently complex programs
Levels of Testing

- **Unit testing**: checking the smallest testable piece
  - A method or class
- **Integration testing**: The interactions among units
- **System testing**: testing the program in context
- **Acceptance testing**: system testing intended to show that the program meets its functional requirements
Some Types of Testing

- **Black-box testing:**
  - Tests item based *only* on its interfaces and functional requirements
  - Assumes no knowledge of internals
- **White-box testing:**
  - Tests *with* knowledge of internal structure
Preparing to Test

- Develop **test plan early**, in the design phase
  - **How** to test the software
  - **When** to do the tests
  - **Who** will do the testing
  - **What** test data to use
- Early test plan allows testing **during** design & coding
- Good programmer practices **defensive programming**
  - Includes code to detect unexpected or invalid data
Testing Tips for Program Systems

• Program systems contain collections of classes, each with several methods

• A method specification should document
  • Input parameters
  • Expected results

• Carefully document (with javadoc, etc.):
  • Each method parameter
  • Each class attribute (instance and static variable)
  • *As you write the code!*
Testing Tips for Program Systems (2)

**Trace execution** by displaying method name as you enter a method:

```java
public static final boolean TRACING = true;
...
public int computeWeight (...) {
    if (TRACING) {
        trace.printf("Entering computeWeight");
    }
    ...
}
```
Display values of all input parameters on entry:

```java
public int computeWeight (float volume,
                        float density) {

    if (TRACING) {
        trace.printf(“Entering computeWeight”);
        trace.printf(“volume = %f, “, volume);
        trace.printf(“density = %f%n”, density);
    }

    ...
}
```
Testing Tips for Program Systems (4)

• Display values of any class attributes (instance and static variables) accessed by the method

• Display values of all method outputs at point of return from a method

• Plan for testing as you write each module,
  • Not after the fact!
Developing Test Data

- **Specify test data** during analysis and design
  - For each level of testing: unit, integration, and system
- **Black-box** testing: unit inputs $\Rightarrow$ outputs
  - Check all *expected* inputs
  - Check *unanticipated* data
- **White-box** testing: exercise all code paths
  - Different tests to make each if test (etc.) true and false
  - Called *coverage*
Developing Test Data (2)

• Helpful to do both black- and white-box testing

• **Black-box** tests can be developed *early* since they have to do with the unit *specification*

• **White-box** tests are developed with detailed design or implementation: *need code structure*
Testing Boundary Conditions

• Exercise *all paths* for
  • Hand-tracing in a structured walkthrough
  • Performing white-box testing
• Must check special cases: *boundary conditions*
• Examples:
  • Loop executes 0 times, 1 time, all the way to the end
  • Item not found
Who does the testing?

• Normally testing is done by
  • The programmer
  • Team members who did not code the module
  • Final users of the product
• Programmers often blind to their own oversights
• Companies may have quality assurance groups
• **Extreme programming:** programmers paired
  • One writes the *code*
  • The other writes the *tests*
Stubs for Testing

- Hard to test a method or class that interacts with other methods or classes
- A **stub** stands in for a method not yet available
- The stub:
  - Has the same header as the method it replaces
  - Body only displays a message that it was called
- Sometimes you need to *synthesize* a reasonable facsimile of a result, for the caller to continue
Drivers

A **driver program**:  
- Declares necessary instances and variables  
- Provides values for method inputs  
- Calls the method  
- Displays values of method outputs  

- A **main** method in a class can serve as a driver to test the class’s methods
Regression Testing

• Once code has passed all initial tests, it is important to *continue to test regularly*
• Environment and other changes ⇒ “*software rot*”
• A **regression test** is designed to:
  • Catch any “regression” or decay in the software
  • Insure old functionality works in face of enhancement
  • Alert earlier to any issues arising from other changes
• Regression testing eased by a **testing framework**
Using a Testing Framework

**Testing framework:** software that facilitates:

- *Writing* test cases
- *Organizing* the test cases into test suites
- *Running* the test suites
- *Reporting* the results
JUnit

• A Java testing framework
• Open-source product
• Can be used stand-alone or with an IDE
• Available from junit.org
JUnit Example

import junit.framework.*;
public class TestDirectoryEntry
    extends TestCase {
    private DirectoryEntry tom;
    private DirectoryEntry dick;
    private DirectoryEntry tom2;

    public void setUp () {
        tom  = new DirectoryEntry("Tom", "...");
        dick = new DirectoryEntry("Dick", "...");
        tom2 = new DirectoryEntry("Tom", "...");
    }
JUnit Example (2)

```java
public void testTomCreate () {
    assertEquals(tom.getName(), "Tom");
    assertEquals(tom.getNumber(), "...");
}

public void testTomEqualsDick () {
    assertFalse(tom.equals(dick));
    assertFalse(dick.equals(tom));
}
```
JUnit Example (3)

```java
public void testTomEqualsTom () {
    assertTrue(tom.equals(tom));
    assertTrue(tom.equals(tom2));
    assertTrue(tom2.equals(tom));
}

public void testSetNumber () {
    dick.setNumber(tom.getNumber());
    assertEquals(tom.getNumber(),dick.getNumber());
}
```
Integration Testing

- Larger components: *collection of classes*
- Done with smaller collection, then larger ones
- Drive with *use cases*: scenarios with
  - Sample user inputs
  - Expected outputs
  - Can be challenging to automate
Debugging a Program

**Debugging:** the major activity during the testing phase

- *Testing* determines that there *is* an error
- *Debugging* determines the *cause*
- Debugging is like detective work: logical deduction
  - *Inspect all program output* carefully
  - *Insert additional output statements* to find out more
  - *Use breakpoints* to examine world ... at *carefully* selected points
Using a Debugger

- **Debuggers** often are included with IDEs
- Debugger supports *incremental* program execution
- *Single-step execution* provides increments as small as one program statement (or even one instruction)
- **Breakpoints** traverse larger portions of code at once
- Details depend on the specific IDE

**Key to debugging:** Think first! Think a lot!

- Also: try to split possible error sources *in half* with each investigation
Reasoning about Programs: Assertions and Loop Invariants

• **Assertions:**
  • *Logical statements* about program state
  • Claimed to be *true*
  • At a particular *point in the program*
  • Written as a *comment*, OR use *assert* statement

• *Preconditions* and *postconditions* are assertions

• *Loop invariants* are also assertions
Reasoning about Programs: Loop Invariants

A loop invariant:
- Helps prove that a loop meets it specification
- Is true before loop begins
- Is true at the beginning of each iteration
- Is true just after loop exit

Example: Sorting an array of n elements
Sorted(i): Array elements j, for 0 ≤ j < i, are sorted
Beginning: Sorted(0) is (trivially) true
Middle: We insure initial portion sorted as we increase i
End: Sorted(n): All elements 0 ≤ j < n are sorted
Efficiency of Algorithms

**Question:** How can we characterize the performance of an algorithm ...  
• Without regard to a specific computer?  
• Without regard to a specific language?  
• Over a wide range of inputs?  

**Desire:** Function that describes *execution time* in terms of *input size*  
• Other measures might be memory needed, etc.
The “Order” of Performance: (Big) O

• Basic idea:
  1. Ignore constant factor: computer and language implementation details affect that: go for fundamental rate of increase with problem size.
  2. Consider fastest growing term: Eventually, for large problems, it will dominate.

• Value: Compares fundamental performance difference of algorithms

• Caveat: For smaller problems, big-O worse performer may actually do better
\[ T(n) = O(f(n)) \]

- \( T(n) \) = time for algorithm on input size \( n \)
- \( f(n) \) = a simpler function that grows at about the same rate

**Example:** \( T(n) = 3n^2+5n-17 = O(n^2) \)
  - \( f(n) \) has faster growing term
  - no extra leading constant in \( f(n) \)
T(n) = O(f(n)) Defined

1. ∃n₀ and
2. ∃c such that

If n > n₀ then c·f(n) ≥ T(n)

Example: T(n) = 3n²+5n-17
Pick c = 4, say; need 4n₀² > 3n₀²+5n₀-17
n₀² > 5n₀-17, for which n₀ = 5 will do.
Efficiency of Algorithms (continued)

Figure 2.12
$3n^2$ vs. $n^2 + 5n + 25$
Efficiency of Algorithms (continued)

**FIGURE 2.13**

$1.5n^2$ versus $1.5n^2 - 1.5n$
### Efficiency of Algorithms (continued)

#### Table 2.5
Symbols Used in Quantifying Software Performance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n)$</td>
<td>The time that a method or program takes as a function of the number of inputs, $n$. We may not be able to exactly measure or determine this.</td>
</tr>
<tr>
<td>$f(n)$</td>
<td>Any function of $n$. Generally $f(n)$ will represent a simpler function than $T(n)$, for example, $n^2$ rather than $1.5n^2 - 1.5n$.</td>
</tr>
<tr>
<td>$O(f(n))$</td>
<td>Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.</td>
</tr>
</tbody>
</table>
Efficiency of Algorithms (continued)

**TABLE 2.6**
Common Growth Rates

<table>
<thead>
<tr>
<th>Big-O</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Log-linear</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>Factorial</td>
</tr>
</tbody>
</table>
Efficiency of Algorithms (continued)
public static int find (int[] x, int val) {
    for (int i = 0; i < x.length; i++) {
        if (x[i] == val) {
            return i;
        }
    }
    return -1;  // not found
}

Letting n be $x.length$:
Average iterations if found $= (1+...+n)/n = (n+1)/2 = O(n)$
Iterations if not found $= n = O(n)$
Hence this is called linear search.
Efficiency Examples (2)

```java
public static boolean allDifferent (int[] x, int[] y) {
    for (int i = 0; i < x.length; i++) {
        if (find(y, x[i]) != -1)
            return false;
    }
    return true;  // no x element found in y
}
```

Letting m be `x.length` and n be `y.length`:

Time if all different = \( O(m \cdot n) = m \cdot \text{cost of search}(n) \)
Efficiency Examples (3)

public static boolean unique (int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x.length; j++) {
            if (i != j && x[i] == x[j])
                return false;
        }
    }
    return true;  // no duplicates in x
}

Letting n be x.length:

Time if unique = n^2 iterations = O(n^2)
Efficiency Examples (4)

```java
public static boolean unique (int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = i+1; j < x.length; j++) {
            if (i != j && x[i] == x[j])
                return false;
        }
    }
    return true;  // no duplicates in x
}
```

Letting \( n \) be \( x\.length \):

Time if unique = \((n-1)+(n-2)+...+2+1\) iterations = \(n(n-1)/2\) iterations = \(O(n^2)\) still ... only factor of 2 better
Efficiency Examples (5)

```c
for (int i = 1; i < n; i *= 2) {
    do something with x[i]
}
```

Sequence is 1, 2, 4, 8, ..., \(\sim n\).

Number of iterations = \(\log_2 n = \log n\).

Computer scientists generally use base 2 for \(\log\), since that matches with number of \textit{bits}, etc.

Also \(O(\log_b n) = O(\log_2 n)\) since change of base just multiples by a constant: \(\log_2 n = \log_b n / \log_b 2\)
Chessboard Puzzle

Payment scheme #1: $1 on first square, $2 on second, $3 on third, ..., $64 on 64th.

Payment scheme #2: 1¢ on first square, 2¢ on second, 4¢ on third, 8¢ on fourth, etc.

Which is best?
Chessboard Puzzle Analyzed

**Payment scheme #1:** Total = $1+$2+$3+...+$64 = $64 \times 65/2 = $1755

**Payment scheme #2:** $1\text{¢}+2\text{¢}+4\text{¢}+...+2^{63}\text{¢} = 2^{64}-1\text{¢} = $184,467,440,737 trillion

Many cryptographic schemes require $O(2^n)$ work to break a key of length $n$ bits. A key of length $n=40$ is perhaps breakable, but one with $n=100$ is not.