CS-101 – WEEK 03 – DEBUGGING AND TESTING

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

Introduction

Debugging, Debugging vs Testing

Program Errors

Compile-Time and Run-Time Errors, Syntax Errors, Logic Errors

Interface Errors, Resource Errors, Arithmetic Errors, Exceptions, Most Common Errors

Debugging Programs

Debugging Tactics, Print Debugging, Interactive Debugging

Debug-Mode vs Release-Mode, Breakpoints, Inspecting Variables, Assertions

Testing Programs

Testing Levels, General Basic Testing, Testing Conditionals, Testing Iterations

Appendices

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- “Java Illuminated (5th Ed.)”, chapter 2, pp. 26-29. *

Additional Resources:

- “Java Illuminated (5th Ed.)”, chapter 5, pp. 219-220.
- “Java Illuminated (5th Ed.)”, chapter 6, pp. 293-294 and pp. 315-317.
- “Java Illuminated (5th Ed.)”, chapter 11, pp. 692-698 and pp. 705-711 and pp. 756-760.
- “Java Illuminated (5th Ed.)”: chapter 14, pp. 1030-1031.

INTRODUCTION

In coding a program, a software developer can encounter different types of programming errors, that could be challenging to fix and sometimes even difficult to detect.

Detecting errors in the code is usually referred as "testing a program", whereas fixing programming errors/bugs in usually called "debugging a program".

Depending on the error type and cause, a programmer has different tools and techniques to detect and then fix the error.

This document includes:

- An overview of the main programming errors.
- A description of the fundamental debugging techniques and tools.
- A summary of standard testing techniques.

Note: Debugging is discussed before testing, because the focus here is on the fixing of errors that are easily detected (with no or minimal testing).

DEBUGGING

In software development, some errors will become apparent without any testing.

Usually, these errors are automatically detected by the compiler, or they become clear during the initial run of a program prototype (not ready yet for release).

So, for these errors, no explicit testing is needed and the developer will have to simply focus on the debugging (identifying the error cause, and fixing the bug).

This is the main reason why, at this time, we focus on debugging techniques.

Later in this course, we will discuss different testing techniques (to detect errors that were not detected during software development).

DEBUGGING VS TESTING

This table shows a summary of the differences between testing and debugging.

TESTING	DEBUGGING
The process to find errors/bugs.	The process to fix errors/bugs found during testing.
The focus is on the detection of errors.	The focus is on the problem-solving of errors.
Done by testers.	Done by programmers.
Can be done without program design knowledge.	Cannot be done without program design knowledge.
Can be done internally or externally.	Can only be done internally.
Can be manual or automated.	Can only be manual. Cannot be automated.
Based on different levels of testing.	Based on different types of errors.
Part of Software Development Life Cycle (SDLC).	Not part of SDLC (just a consequence of testing).
Includes validation and verification.	Matching symptoms-causes, then fixing errors.
Starts after code is implemented.	Starts after a test case is completed.

PROGRAM ERRORS

Program error: A bug (error) in the code, resulting in incorrect results or poor performance.

Depending on their "location", programming errors can be classified in:

- Compile-time errors.
- Run-time errors.

Depending on their "cause", programming errors can be classified in:

- Syntax errors.
- Logic errors.
- Interface errors.
- Resource errors.
- Arithmetic errors.

COMPILE-TIME ERRORS

Compile-time error: Aka compilation error, this is an error occurring during compilation.

The compiler detects an issue, prints an error-reporting message on console, and compilation fails (no generation of executable code).

Example: This is an example of a code line generating a compile-time error.

```
int a = 1.3F; // Code line with a compile-time error!
```

And its relative error-reporting message displayed by the compiler on the console.

```
Error: incompatible types: possible lossy conversion from float to int.
```

Note: Usually, error-reporting messages include details on error cause and error location (code file name, and code line). Read the message, and consider that the compiler could be incorrect on those details (but not on the existence of an error).

RUN-TIME ERRORS

Run-time error: This is an error occurring during the running of a program.

Depending on the situation, the detection/reporting is done by the interpreter (via exceptions), the OS, or the programmer (via testing).

In these cases (run-time errors), the compilation is successful (executable code generated), but the executable still has issues.

Example: This is an example of a code line generating a run-time error.

```
int[] arrayA = new int[5]; // Creation of array with 5 cells/items.
int b = arrayA[10]; // Code line causing a run-time error!
```

This specific run-time error is detected by the interpreter (JVM) that signals it by using an exception (in this case unhandled, and displayed on the console).

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 10 out of bounds for length 5.
```

SYNTAX ERRORS

Syntax error: This is an error in the syntax (grammar) of a statement in the code (accordingly to the specifications of the programming language used).

These errors are (almost) always compile-time errors.

Example: This is an example of a code line generating a syntax error (compile-time error).

```
int a = 123 // Code line with a syntax error (compile-time error)!
```

This specific syntax error is also a compile-time error, and this is its relative error-reporting message displayed by the compiler on the console.

```
Error: ';' expected.
```

Note: Syntax errors are always compile-time errors in compiled programming languages.

Syntax errors are always run-time errors in interpreted programming languages.

LOGIC ERRORS

Logic error: This is an error in the logic of the program, that is the code syntax is correct but the computation is not (incorrect results, control flow, etc.).

Usually, these errors are run-time errors or they are not automatically detected.
Some logic errors can be automatically detected by the interpreter/OS.
Other logic errors can be detected only by the programmer via testing.

Example: This is an example of a code line generating a logic error (run-time error).

```
boolean variableAEqualToZero = false;
// Incorrect conditional statement causing a logic error!
if( a == 0 ) {
   variableAEqualToZero = false; }
else {
   variableAEqualToZero = true; }
```

Note: Usually, logic errors are the most difficult to detect, especially if rare.

INTERFACE ERRORS

Interface error: This is an error in the interfacing (internal/external input/output communication) of the program (or part of the program). For example: the inputs of the program do not follow the required standards.

Usually, these errors are run-time errors.

Example: This is an example of a code line generating an interface error (run-time error).

```
// Code line with interface error if "func" expect 1st parameter to be non-zero!
int a = func(0, 123);
```

Note: Unless handled correctly, interface errors are difficult to locate (not easy to find the code file and/or code line where the issue is).

RESOURCE ERRORS

Resource error: This is an error in the resource allocation (processor, memory, etc.) of the program. For example: failure to free the memory of unused variables.

Usually, these errors are run-time errors (crashes) or they are not automatically detected (memory leaks, infinite loops).

Example: This is an example of a code line generating a resource error (run-time error).

```
// Code line with a resource error:
// infinite loop causing array to exhaust memory!
int[] arrayA = new int[1];
while(true) {
   arrayA = new int[ arrayA.length + 1 ];
}
```

Note: Usually, resource errors are difficult to detect and fix, because they may require monitoring memory/processor load and extensive testing.

ARITHMETIC ERRORS

Arithmetic error: This is an error in the mathematic computation of the program. For example: a division by zero.

Usually, these errors are run-time errors.

Example: This is an example of a code line generating a resource error (run-time error).

```
// Code line with an arithmetic error (integer division by 0 if c == 0)!
int a = b/c;
```

Note: Usually, arithmetic errors are detected by testing (special cases, etc.).

Fixing arithmetic errors can be easy or difficult, depending on the math involved.

EXCEPTIONS

Exceptions will be discussed in detail later in this course, this is just a brief introduction.

Exception: A signal that an unplanned event occurred during the execution (run-time) of a program, disrupting the normal program operation.

Exceptions are generated (thrown) by the code itself (the programmer).

Exceptions can be classified as critical (checked) or non-critical (unchecked), depending on the "unplanned event" they are related to.

Exception handling: The process of responding to the occurrence of exceptions by coding special processing including ad-hoc programming statements (try-catch blocks).

MOST COMMON ERRORS

This is a short list of the most common errors in programming:

- Misunderstanding default values.
- Misunderstanding scope of variables.
- Misunderstanding references.
- Using incorrect references.
- Using incorrect comparison methods.
- Using incorrect array indices.
- Using incorrect iterations.
- Neglecting to free resources.
- Neglecting exception handling.
- Neglecting compiler messages.

DEBUGGING PROGRAMS

Debugging: In programming, debugging is the process of finding and resolving bugs (errors or problems) in a software system (code, external devices, etc.).

If you have programmed, you have debugged.

Usually, any program will have some bugs in its first version.

Debugging (as well as programming) is an art, so it is difficult to teach how to master it.

Note: We will discuss only the debugging of code, focusing on code correctness.

DEBUGGING TACTICS

In general, you can use different debugging tactics depending on the situation and goal:

- Print debugging and interactive debugging.
- Control flow analysis.
- Unit/integration/system/acceptance/alpha/beta testing (see testing levels).
- Log file analysis, memory dumps, and monitoring the application/system.
- Profiling, and specific debugger features.

Note: We will focus only on simple debugging techniques (print and interactive debugging). You will see advanced debugging tactics and tools later in other programming courses.

PRINT DEBUGGING

Print debugging: Investigating bugs by adding temporary print statements in the code (to check the values of variables, or the control flow of the program).

Print debugging is the simplest strategy to investigate a bug.

So, usually, it is the strategy of choice for beginner programmers.

Print debugging is time-consuming (in respect to interactive debugging), but in some situation it may be the only debugging strategy available.

Example: A simple example of print debugging.

```java
System.out.println("Before init var a."); // Print debug.
int a = 3 * 3;
System.out.println("After init var a, value: " + a); // Print debug.
```

INTERACTIVE DEBUGGING

Interactive debugging: Investigating bugs using a special software tool called debugger.

Debugger: A software tool that can inspect other processes (e.g., your code in execution) and view their internal state (e.g., the variables in your code at runtime).

For a debugger to work, the code must be compiled/run in debug-mode.

Some of the main functionalities provided by a debugger are:

- Execute code line-by-line (breakpoints).
- Pause-resume code execution at specific code lines (breakpoints).
- Inspect variables during code execution (watch panel).

Note: We will discuss only the main general debugging functionalities (always available). In terms of debugging interface, we will discuss only the jGRASP interface.

DEBUG-MODE VS RELEASE-MODE

Most compilers/IDEs (for different programming languages) provide 2 different modalities to compile/run your code: debug-mode, and release-mode.

Debug-mode: Code compiled/run in debug-mode enables the use of a debugger (providing special functionalities to debug your program).

Code compiled/run in debug-mode includes additional information to facilitate the debugger, so it is not optimized for performance.

Release-mode: Code compiled/run in release-mode is considered production-ready (with no known bugs), and usually it is optimized for performance.

Code compiled/run in release-mode does not allow the use of a debugger.

BREAKPOINTS

Usually, when you run a program, the execution starts at the beginning of the code (entry point) and run until the last instruction (if no run-time error occurs).

Breakpoint: A special mark on a code line, that tells the debugger to pause the execution in debug-mode at that code line (execution is paused before executing that line).

Breakpoints are ignored if a program is run in release-mode.

Once the execution (in debug-mode) is paused at a breakpoint location, the programmer can:

- Check the status of variables (watch panel).
- Execute only the current code line (stepping over), pausing again at the next code line.
- "Enter" a function call (stepping in), executing the code of the function line-by-line.
- Resume the execution, until completion or the next breakpoint.

INSPECTING VARIABLES

Inspecting ("watching") variables at run-time is an important task in interactive debugging.

This task relies on the functionalities provided by a debugger/IDE.

Usually, each debugger/IDE provides its own interface to inspect variables.

Example: In jGRASP, while running the code in debug-mode, the programmer can use the "Debug" side panel ("Variables" tab) to inspect the variables currently active.

Additionally, jGRASP marks in red color the last variable modified.

Finally, jGRASP integrates 2D visualization/animation functionalities to illustrate data structures (both array-based and reference-based).

INTERACTIVE DEBUGGING SESSION

Interactive debugging is based on using the functionalities of a debugger over and over again: starting from the detection of a bug, until the code works.

This is a brief list of steps of a typical interactive debugging session:

- Detection of a bug in the code, with a decent understanding of its location (code line).
- If the bug needs to be investigated, place a breakpoint at the target code line.
- Run the code in debug-mode.
- Once the debug-mode execution is paused at the breakpoint, start your investigation.

The investigation of a bug is usually focused on (1) fixing the bug or (2) localizing the bug.

In general, investigating a bug involves:

- Checking the values of some variables.
- Checking the control flow of your program.

ASSERTIONS

Assertion: An assertion is a programming statement that enables the programmer to test an assumption about the program.

For example: in coding the computation of the speed of an object, the programmer could assert that the speed should be less than the speed of light.

In practice, an assertion is a boolean expression that the programmer believes to be true at the code line where the assertion is.

If assertion passes, nothing happens, and code execution proceeds as normal. If assertion fails, an error is signaled.

Example: This is a Java assertion to check that a month is in a valid range (1-12).

```java
int currentMonth = 2; // Variable storing a month.
// Assertion checking that the month is in valid range.
assert ( ( currentMonth > 0 ) && ( currentMonth < 13 ) );
```

ASSERTIONS (2)

Note: Remember that in most programming languages, assertions are limited to debug-mode or disabled by default (they can be enabled by using compiler settings).

So, never assume that assertions will be executed (check compiler settings)!

Example: In Java, assertions are disabled by default.

In jGRASP, enable assertions using menu "Build", and selecting "Enable Assertions".

Note: It is a good programming practice to use assertions when needed, because they are very useful to detect and correct bugs (on computation and control flow).

Additionally, assertions serve to document your code, improving readability and maintainability.

TESTING PROGRAMS

Testing software: In software development, testing is the examination of the results and behavior of a program by validation and verification.

The main goal of testing software is to provide objective and independent information about the quality of a program (correctness, completeness, performance, failure risk, design, etc.).

Usually, testing a program, the programmer cannot prove that the code works properly under all conditions, but only that it does not function properly under specific conditions.

Note: Nowadays, in software development, testing is decoupled by programming (i.e., testers are different than programmers).

Note: Testing results can be used in software development (iterative design) before a program reaches its release version.

TESTING LEVELS

There are multiple ways to test a program.

A general basic testing can/should be done during the development of a program to verify that the code is correct (generating accurate results).

Additionally, there are 4 other advanced main levels of testing:

- **Unit testing:** Verify the functionalities of a "unit" of code (a function, a class, etc.).
- **Integration testing:** Verify the interfaces between software components.
- **System testing:** Verify that a complete software system meets its requirements.
- **Acceptance testing:** Verify that target audience accepts a program (e.g., alpha/beta).

Note: This is just a very brief introduction to testing levels.
There are many more specific testing levels, and several different types of testing.

GENERAL BASIC TESTING

General basic testing: After a program is compiled successfully, it is necessary to verify that the code is correct, that is: checking that the program generates accurate results for any possible input.

Usually, it is not feasible to test a program with all possible inputs.
However, we can perform a reasonable correctness check by focusing the testing on:

- Check program correctness with known inputs (manual solutions comparison, testing boundary conditions/values, etc.).
- Check program correctness with invalid inputs (inputs out-of-range, error messages, automatic adjustment of inputs, etc.).
- Check program correctness assuming it is a "black box" (assuming code unknown, testing based only on specifications).

TESTING CONDITIONALS

The main test to verify the correctness of a conditional (if-then-else) is:

- Check all control flow paths (via print debugging and temporary local variables).

Example: These code samples illustrate situations that may require some testing.

```
int x = 123; // Variable to test all control flow paths.
// Nested conditional (if-then-else).
if( x > 0 ) {
   // This conditional branch is executed when x is positive.
}
else if( x < 0 ) {
   // This conditional branch is executed when x is negative.
}
else {
   // This conditional branch is executed when x is zero.
}
```

TESTING ITERATIONS

The main tests to verify the correctness of an iteration (for, while, do-while) are:

- Check that the iteration performs the correct number of loops (via print debugging).
- Check the situation when the control flow should completely skip the iteration body (loop condition fails immediately).

Example: These code samples illustrate situations that may require some testing.

```
// Iteration (for) header configured to execute 4 iterations (not 5).
for( int i = 1; i < 5; i++ ) {
   // Iteration body.
}
```

```
boolean flag = false; // Variable representing iteration condition.
// Iteration (while) condition fails immediately.
while(flag) {
   // Iteration body.
}
```

APPENDICES

Appendices

- Null Pointer Exception
- Array Index Out of Bound Exception
- Floating-Point Comparison
- Infinite Loop
- No-Op Effect
- Short-Circuit Evaluation
- The Call Stack
- ISTQB Certified Tester Foundation Level
- Unit Test Example