Inf1B
Testing and Debugging

Perdita Stevens
adapting earlier versions by Ewan Klein, Volker Seeker, et al.

School of Informatics
Things will go wrong

There is usually an error in your code somewhere.

Types of Errors

Ordered by difficulty to detect and fix them.

- Syntax Errors
- Runtime Errors
- Logical Errors
Syntax Errors

Comparable to a spelling mistake in a text.

```
int value = 5;
if (value < 10
    System.out.println("Here we are.");
```
Syntax Errors

Comparable to a spelling mistake in a text.

This is a speling mistake!

```java
int value = 5;
if (value < 10)
    System.out.println("Here we are.");
```

An IDE can help you detect them.
Syntax Errors

Syntax errors are detected at compile time.

Compiler Output

Main.java:5: error: '}' expected
   if (value < 10
        ~
Main.java:6: error: ';' expected
    System.out.println("Here we are.")
        ~
2 errors
Syntax Errors

Not always easy to identify despite compiler and IDE help.

```java
public class Main {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println(add(5, 5));
    }
}
```

Compiler Output

```
Main.java:5: error: illegal start of expression
    public static void main(String[] args) {
^ 1 error
```
Runtime Errors

Comparable to a grammar mistake in a text.

```java
int[] arr = { 1, 2, 3, 4 }; System.out.println(arr[4]);
```
Runtime Errors

Comparable to a grammar mistake in a text.

```java
int[] arr = { 1, 2, 3, 4 };  
System.out.println(arr[4]);
```

Compiler and IDE are unable to detect them.
Runtime Errors

The Java Runtime will detect them and crash your program.

```java
int[] arr = { 1, 2, 3, 4 }; System.out.println(arr[4]);
```

Runtime Output

```java
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException:
    Index 4 out of bounds for length 4
at Main.main(Main.java:5)
```
Logical Errors

Comparable to an incorrect or unintended statement in a text.

The swan is an orange bird.

```java
public static int add(int a, int b) {
    return a - b;
}
```
Logical Errors

Comparable to an incorrect or unintended statement in a text.

The swan is an orange bird.

```java
public static int add(int a, int b) {
    return a - b;
}
```

Neither compiler, nor IDE or Java Runtime can detect them.
Logical Errors

You need to test your code to catch them.

```java
public static int add(int a, int b) {
    return a - b;
}

public static void main(String[] args) {
    if (add(5, 5) != 10)
        System.out.println("Unexpected sum!");
}
```
Types of Errors

Ordered by difficulty to detect and fix them.

- Syntax Errors
- Runtime Errors
- Logical Errors
Types of Errors

Ordered by difficulty to detect and fix them.

- Syntax Errors *Caught at compile time*
- Runtime Errors
- Logical Errors
Types of Errors

Ordered by difficulty to detect and fix them.

- Syntax Errors **Caught at compile time**
- Runtime Errors **Caught at runtime**
- Logical Errors
Types of Errors

Ordered by difficulty to detect and fix them.

- Syntax Errors **Caught at compile time**
- Runtime Errors **Caught at runtime**
- Logical Errors **Caught via testing**
Types of Errors

Ordered by difficulty to detect and fix them.

▶ Syntax Errors **Caught at compile time**
▶ Runtime Errors **Caught at runtime**
▶ Logical Errors **Caught via testing**

**NB** Since tests execute your code, they will also catch runtime errors.
Let’s hunt some bugs!

Created by Vectors Market from Noun Project
Let’s hunt some bugs!

1. Testing  *detect the errors*
2. Debugging  *find and fix the errors*
Testing
Regression Testing

Regression:
"when you fix one bug, you introduce several newer bugs."

Source: https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/
Test Driven Development

Source: https://dzone.com/articles/what-is-refactoring
Implement a utility class with calculator functionality.
How would you test the functionality of a class?

Demo
Main methods can be used to quickly evaluate the functionality of your code.
Main Method as Test Client

Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:
Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:

- Using console output to evaluate test results requires manual effort and is error prone for more complex tests.
Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:

- Using console output to evaluate test results requires manual effort and is error prone for more complex tests

→ use assertions instead!
Automatic evaluation with assertions

Demo
Main Method as Test Client

Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:

- Using console output to evaluate test results requires manual effort and is error prone for more complex tests
  
  → use assertions instead!
Main Method as Test Client

Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:

- Using console output to evaluate test results requires manual effort and is error prone for more complex tests → use assertions instead!
- tests are unorganised, no easy way to test only certain methods
Main Method as Test Client

Main methods can be used to quickly evaluate the functionality of your code.

They have, however, a few drawbacks:

- Using console output to evaluate test results requires manual effort and is error prone for more complex tests
  → use assertions instead!
- tests are unorganised, no easy way to test only certain methods
  → use a test framework instead!
Organising Tests with a Test Framework

Demo
Testing Strategies

- test for regular use cases
- test for corner cases
- test for invalid input (how should it be handled?)
- positive testing vs. negative testing
Debugging
Something is wrong with this array rotation code.

```java
int[] arr = { 1, 2, 3, 4, 5 };                      
int tmp = arr[arr.length - 1];                     
for (int i = 0; i < arr.length - 1; i++)          
{
    arr[i + 1] = arr[i];                           
}
arr[0] = tmp;
```

Let’s find out what without the help of machines.
Logging

With Compiler and Runtime, we can use a logging approach.

```java
int[] arr = {1, 2, 3, 4, 5};
int tmp = arr[arr.length - 1];
for (int i = 0; i < arr.length - 1; i++) {
    arr[i + 1] = arr[i];
    System.out.println(Arrays.toString(arr));
}
arr[0] = tmp;
System.out.println(Arrays.toString(arr));
```

Output

```
[1, 1, 3, 4, 5]
[1, 1, 1, 4, 5]
[1, 1, 1, 1, 5]
[1, 1, 1, 1, 1]
[5, 1, 1, 1, 1]
```
Using a Debugger

With the help of a debugger, we can get a lot of information without much effort from our side.

```
int[] arr = { 1, 2, 3, 4, 5 };  arr: {1, 1, 1, 4, 5}
int tmp = arr[arr.length - 1];  tmp: 5
for (int i = 0; i < arr.length - 1; i++) {
    arr[i + 1] = arr[i];  arr: {1, 1, 1, 4, 5}
    i: 2
}
arr[0] = tmp;
System.out.println(Arrays.toString(arr));
```
Debugging Strategies

- Manual Walk Through
- Logging
- Debugger
Bug Hunting

1. Testing *detect the errors*
2. Debugging *find and fix the errors*
Bug Hunting

0. Write Robust and Maintainable Code
   avoid errors in the first place

1. Testing detect the errors
2. Debugging find and fix the errors
Error Handling
Handling Invalid Input

Given a function that generates a sequence of numbers
What could go wrong here?

```java
public static int[] sequence(int start, int end) {
    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

Start could be smaller than end! How could we handle this best?
Handling Invalid Input

Given a function that generates a sequence of numbers
What could go wrong here?

```java
public static int[] sequence(int start, int end) {
    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

Start could be smaller than end!
Handling Invalid Input

Given a function that generates a sequence of numbers
What could go wrong here?

```
public static int[] sequence(int start, int end) {
    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

Start could be smaller than end!
How could we handle this best?
Handling Invalid Input

Make a note in the function documentation.

```java
/** Start must always be smaller or equal to end! */
public static int[] sequence(int start, int end) {
    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```
Handling Invalid Input

Make a note in the function documentation.

```java
/** Start must always be smaller or equal to end! */
public static int[] sequence(int start, int end) {
    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

Helpful but not a good way to enforce rules.
Handling Invalid Input

Add a check and print an error message.

```java
/** Start must always be smaller or equal to end */
public static int[] sequence(int start, int end) {
    if (start > end)
        System.err.println("ERROR: Start must be smaller end!");

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```
Handling Invalid Input

Add a check and print an error message.

```java
/** Start must always be smaller or equal to end */
public static int[] sequence(int start, int end) {
    if (start > end)
        System.err.println("ERROR: Start must be smaller than end!");

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

More helpful, but this will still crash.
Handling Invalid Input

For internal code during development, a crash might be sufficient. But you should use an assertion in that case.

```java
/** Start must always be smaller or equal to end */
public static int[] sequence(int start, int end) {
    assert start < end : "Start must be smaller than end."

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```
Handling Invalid Input

For internal code during development, a crash might be sufficient.
But you should use an assertion in that case.

```java
/** Start must always be smaller or equal to end 
! */
public static int[] sequence(int start, int end) {
    assert start < end : "Start\_must\_be\_smaller\_end ".

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

Not enough for publicly exposed function used by others.
/** Start must always be smaller or equal to end!
 * Null will be returned otherwise. */

public static int[] sequence(int start, int end) {
    if (start > end) return null;

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
Handling Invalid Input

Return an error value.

```java
/** Start must always be smaller or equal to end
 * Null will be returned otherwise. */
public static int[] sequence(int start, int end) {
    if (start > end) return null;

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

This will avoid the error and report it to the calling code but it does not always work.
Handling Invalid Input

Return an error value.

```java
public static int sum(int[] data) {
    if (data.length == 0) return ??????

    int result = 0;
    for (int d : data) {
        result += data;
    }

    return data;
}
```

This will avoid the error and report it to the calling code but it does not always work.
Handling Invalid Input

Throw an Exception.

```java
/** Start must always be smaller or equal to end!
 * IllegalArgumentException is thrown otherwise. */
public static int[] sequence(int start, int end) {
    if (start > end)
        throw new IllegalArgumentException("Start must be smaller end.");

    int[] result = new int[end - start];
    int index = 0;
    while (start < end) {
        result[index++] = start++;
    }
    return result;
}
```

This reports the error without contaminating the return value.
There is a short exercise on handling errors in the calling code in the labs.
Handling Invalid Input

**Inf1B Coding Conventions:**
For **private** methods:
Use assertions if it helps you during development.

For **public** methods:

▶ Note error handling in the documentation.
▶ Throw `IllegalArgumentException` for illegal arguments.
▶ Throw `NullPointerException` for null arguments.
▶ If explicitly stated: handle via return value.
Summary

- Three types of errors: syntax, runtime and logical
- Three testing strategies: main, assert, unit
- Three debugging strategies: manual, print, debugger
- Three ways for error handling: assert, return, exception
Objects First
Chapter 9 (some *BlueJ* specifics and techniques I have not yet fully taught you but good examples. Feel free to ignore functional bit.)

Java Tutorial
Chapter 10 (Mostly about exceptions and exception handling.)