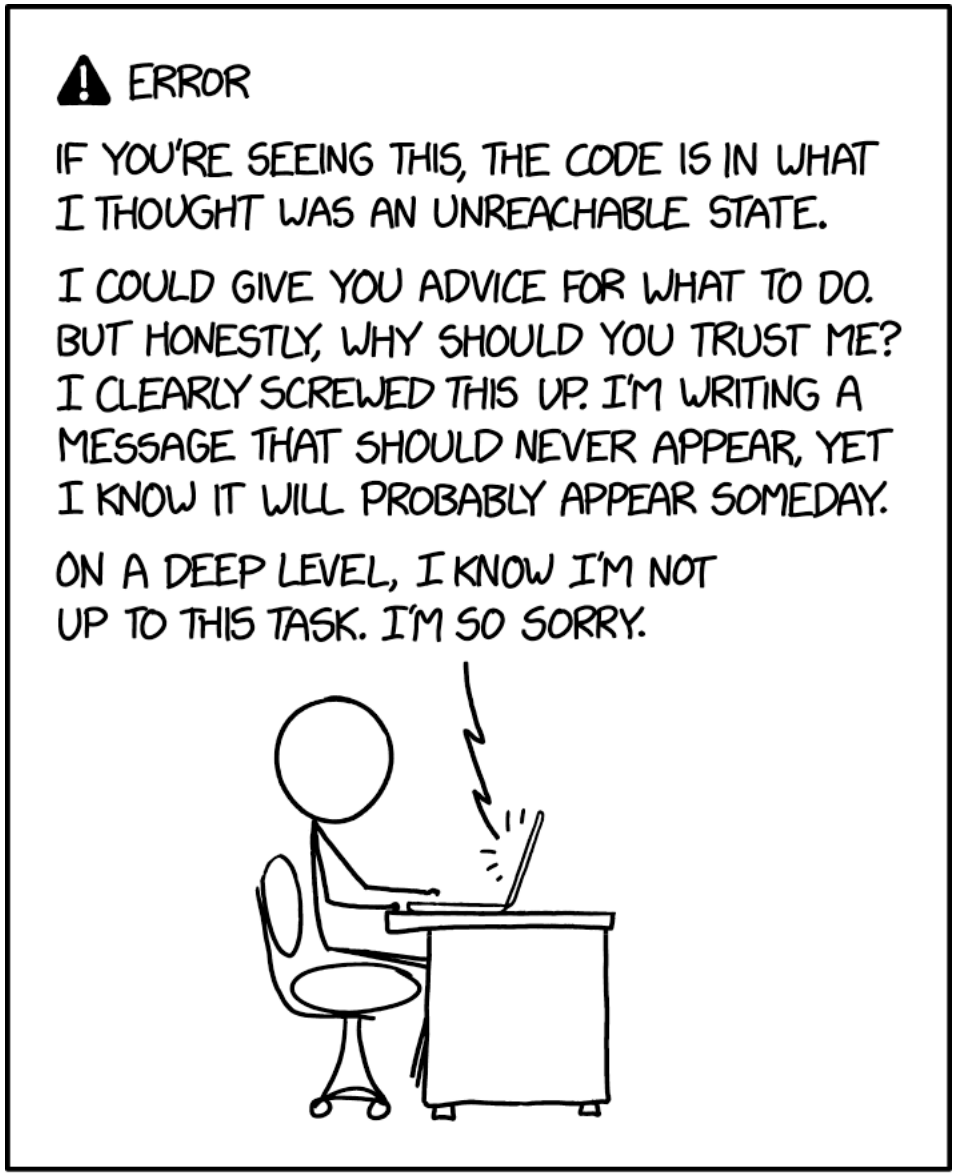


NEVER WRITE ERROR MESSAGES TIRED.

ERROR: We've reached an unreachable state. Anything is possible. The limits were in our heads all along. Follow your dreams.

Defensive Programming

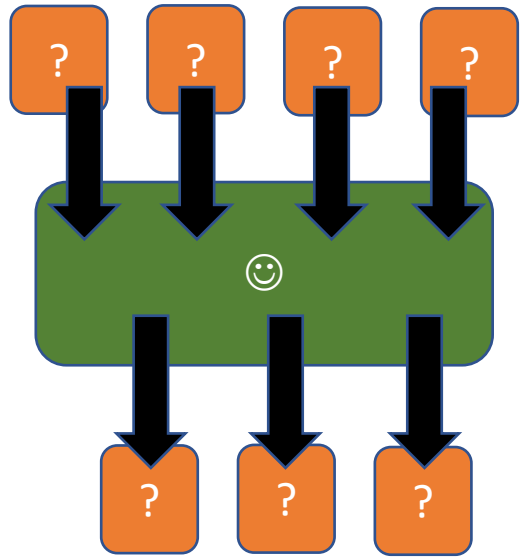
CSCI 2134: Software Development

Agenda

- Lecture Contents
  - Motivation
  - Defensive Programming
  - Assertions
    - Live Demo Code
  - Exceptions
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 8
  - Next Lecture: Chapter 8

What is Defensive Programming?

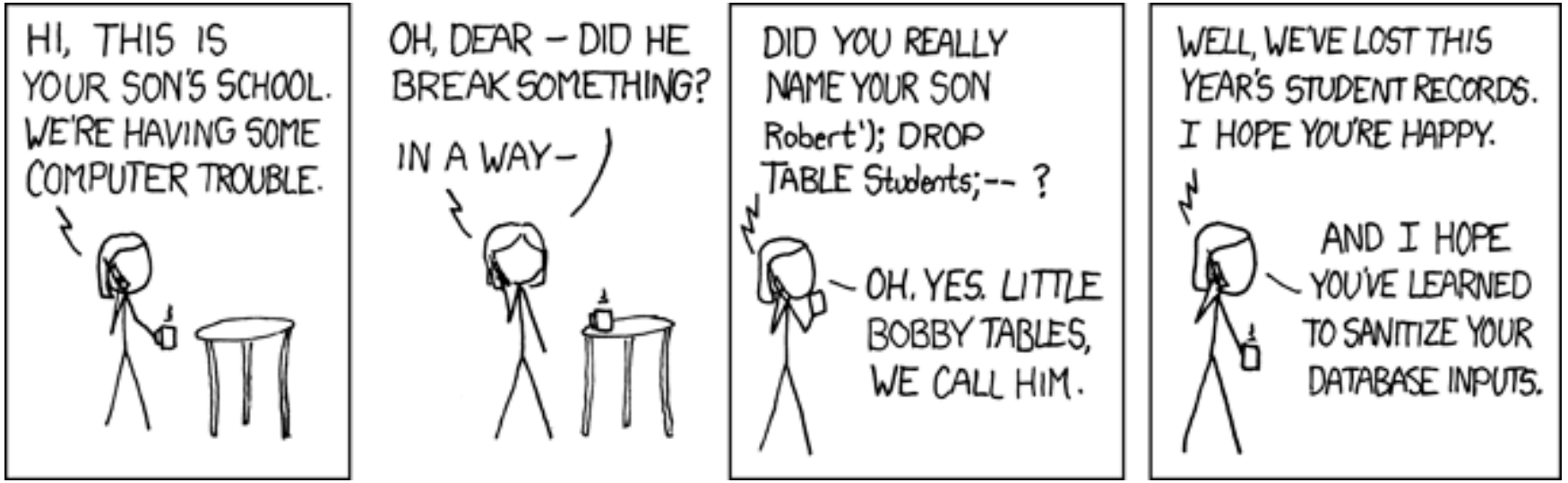
- **Definition**: A programming style that protects your implementation from errors in how
  - Other parts of the program may use your code or methods
  - Your code and methods uses other code
- Characteristics of Defensive Programming:
  - Make as few assumptions as possible
  - Test or validate all assumptions that you make
- Goals of Defensive Programming:
  - **Robustness**: Ensure that your code continues to run no matter what bad information comes its way
  - **Correctness**: Ensure that your code never returns an inaccurate result



The Fundamental Question of Defensive Programming

- **The Question:** How can other code affect my code?
- **The Answer:**

  The data injected into my code.
- If our assumptions about the data are wrong, our program crashes. ☹



Ways in which Data is Injected into Our Code

- Data read in from other sources
  - User input
  - Data bases
  - Files
  - Environment (OS)

```
Scanner stdin = new Scanner(System.in);
…
int code = stdin.nextInt();
…
```

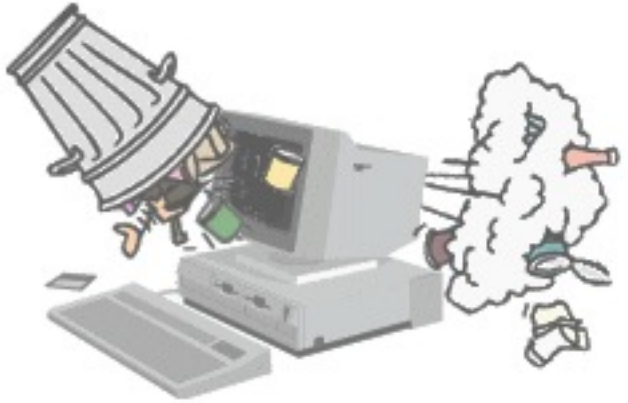
- Parameter values to my methods

```
public int myMethod(int idx, int [] arr) {
  …
  int result = arr[idx];
  …
```

- Return values from the methods my code calls

```
int [] arr = getArray();
…
int result = arr[idx];
…
```

Rules of Defensive Programming



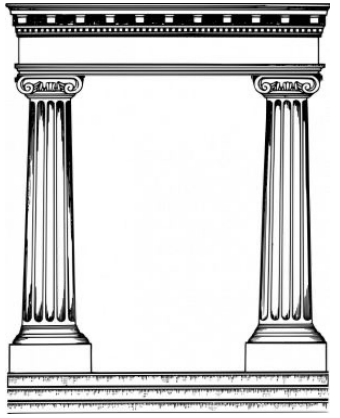
- Don’t propagate the error
  - Avoid “Garbage in, garbage out” situations
  - Example: Calculate GPA of the grades: B, B+, E, A-, C
    Computing this blindly and returning a meaningless result serves no purpose
- Don’t corrupt your own data with bad data

  Example: If you are passed a bad index value, don’t use it!
- Validate all data before using

  Costs:
  - More code and time

    As much as 80% of the code may be for validation and error handling
  - Less efficient code (occasionally)
  - More unit tests ☺

The Pillars of Defensive Programming



- Two parts:
  - **Data validation**: Detecting invalid data or violated assumptions
  - **Error handling**: Dealing with invalid data or violated assumptions
- Data validation involves:
  - Understanding what can go wrong
  - Building tests right into the code
  - Limiting assumptions we are making about the data
- Error handling involves:
  - Understanding the consequences of not proceeding with the data
  - Using appropriate mechanisms to recover from or deal with bad data

What Can Go Wrong with Our Data?

Type of Problem	Ease of Detection	Example
Null reference or pointer	Easy	Self explanatory
Bad reference or pointer	Hard	Uninitialized pointer (in C or C++)
Bad format	Depends on format	Date MM/DD/YY instead of DD/MM/YY
Data not in range	Easy	Index is -1 instead of between 0 and …
Data not one of enumerated values	Easy	Answer is “Maybe” instead of either “Yes" or ”No”
Restricted characters in a string	Easy	The & character in HTML
Size of data too big	Easy	Input line is too big for the buffer
Complex data structures are corrupted	Depends on data structure	Next reference of tail node of a linked list is not null

Observation: Most of these data problems are easy to detect!

- **if** statements
- **assert** statements

Error Handling

- Once we know there is a problem, we need to handle it
- An error occurs when the current computation cannot proceed due to bad data
- Two things occur when an error occurs:
  - **Recovery**: The programs moves to a safe state from which it can proceed
  - **Report**: The program reports that an error has occurred
- Recovery mechanisms:
  - **Abort (Shutdown)**: In some cases, this is the only safe thing to do.
  - **Return to caller**: Error handling is deferred to the code that called this code
  - **Exceptions**: Invoke a specific error handling code
- Reporting mechanisms:
  - **No reporting**: If the error can be corrected, or there is no point to reporting the error
  - **Return error code**: The error is reported to the code that called this code
  - **Logging**: Notice is sent to a file or a terminal for later review
  - **Error message**: A dialog box or console indicating an error has occurred

Assertions: Abort with Error Message

- Assertions are a mechanism for
  - Encoding the assumptions in our code
  - Enforcing these assumptions
  - Aborting and reporting when an assumption is violated
- An **assertion** is additional code that specifies and uses a Boolean condition to represent an assumption
- When the code is executed (with assertions enabled)
  - The assertion checks the condition
  - If the condition evaluates to false,
    - An error message is printed
    - The program is aborted

Assertions in Java

- Assertions in Java have two forms:
- **Simple**: `assert condition;`

  For example: **`assert head != null;`**
- **With Message**: `assert condition : message;`

  For example: **`assert head != null : "List is empty";`**
- In both cases, when the assertion is executed:
  - The condition is evaluated
  - If the condition evaluates to false, a Java *`AssertionError`* exception is thrown
  - This exception typically aborts the program

Example 1: Arrrr (again)

- The code:

```
int max(int [] arr)) {
  // assume arr != null
  // assume arr.length > 0
  int maxValue = arr[0];
  for (int i = 1; i < arr.length; i++) {
    if (arr[i] > maxValue) {
      maxValue = arr[i];
    }
  }
  return maxValue;
}
```

- What is the assumption?
  - array is not null
  - array is not length 0

- The solution with assertions:

```
int max(int [] arr)) {
  assert arr != null;
  assert arr.length > 0;

  int maxValue = arr[0];
  for (int i = 1; i < arr.length; i++) {
    if (arr[i] > maxValue) {
      maxValue = arr[i];
    }
  }
  return maxValue;
}
```

Example 2: Return it (once more)

- The code:

```
arr = list.toArray();
// assume arr.length > 1
if (arr[0].equals(arr[1])) {
  return true;
}
```

- What is the assumption?

  Array returned has at least two elements

- Note: We could also assume arr != null

- The solution with assertions:

```
arr = list.toArray();
assert arr.length > 1;

if (arr[0].equals(arr[1])) {
  return true;
}
```

When to use Assertions?

- During development phase
  - Assertions are not executed in production code
  - Note: In Java, you need to explicitly enable assertions using the -ea switch on the Java VM
  - In IntelliJ follow: *Run → Edit Configurations... → Configuration → VM options* and add *the* `-ea` *switch*
- Only for assumptions that **always** hold true
  - Correct user input is not such an assumption
- When a boolean condition can be used to test the assumption
- To verify preconditions and postconditions of a code block

Aside: Pre- and Post-Conditions

- **Pre-Conditions** are assumptions (conditions) that must hold true at the start of a block of code, in order for the block to execute correctly
- **Post-Conditions** are assumptions (conditions) that should hold true if the pre-conditions were true and the block has executed correctly

```
// preconditions
assert arr != null;
assert arr.length > 0;

int maxValue = arr[0];
for (int i = 1; i < arr.length; i++) {
  if (arr[i] > maxValue) {
    maxValue = arr[i];
  }
}
// postcondition: maxValue >= arr[i] for i = 0 … arr.length;
return maxValue;
```

Cannot easily be implemented as an assertion

Example 3: To err is human

- The code:

```
Scanner scanner =
    new Scanner(System.in);

System.out.print("Enter mailbox #:");
// assume good user input
int mailboxId = scanner.nextInt();
Mailbox mailbox =
    list.get(mailboxId);
...
```

- What is the assumption?
  - User will enter an integer
  - User will enter correct integer

- The solution:

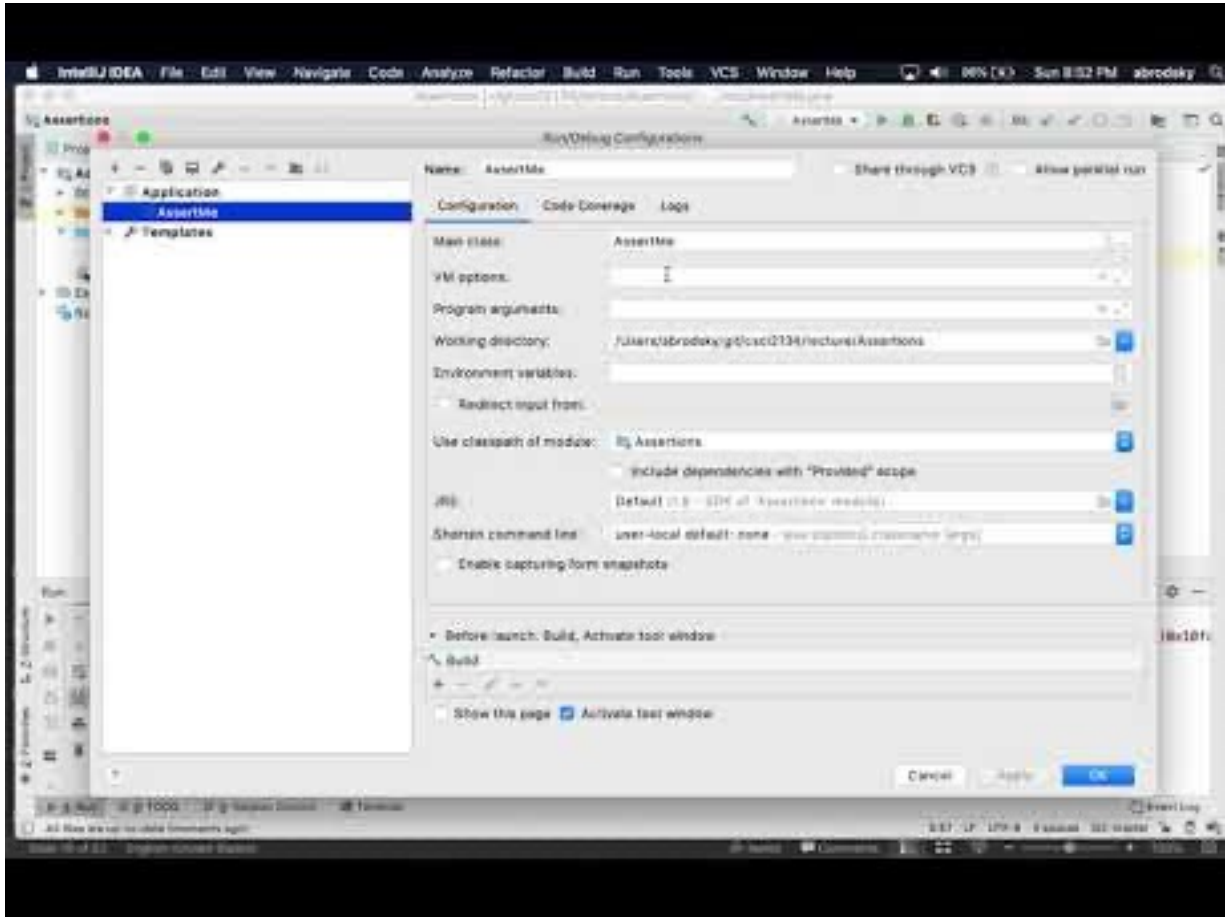
```
Scanner scanner =
    new Scanner(System.in);

System.out.print("Enter mailbox #:");
assert scanner.hasNextInt();
int mailboxId = scanner.nextInt();
assert mailboxId >= 0;
assert mailboxId < list.size();
Mailbox mailbox = list.get(mailboxId);
...
```

- This is not a good idea!
  - Do not make assumptions about the user!
  - Better recovery is needed to deal with bad user input!

Live Demo of Assertions

Code here: https://git.cs.dal.ca/courses/2022-winter/csci-2134/lecture/assertions.git



https://youtu.be/ETC7Hox2jxY

The Problem with Assertions

- Assertions do not allow the program to continue running (no recovery)
- Assertions (typically) do not log or make a record of the problem
- Assertions are used in development, but not in production code

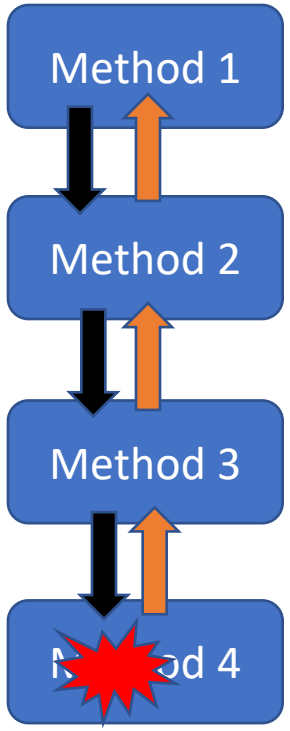
- Ideally:
  - The goal is for the program to deal with the error rather than crash
  - The program needs to be notified that something went wrong and handle the error
- How?
  - Return codes (old-way)
  - Exceptions (standard way)

Return Codes

- **Idea**: the API for our code reserves special values indicating an error
- Examples:
  - In **Matrix.java** `null` is returned if a bad parameter is passed to some of the methods
  - In C, many standard library functions return a negative value if an error occurs
    - Note: C does not have exceptions
- Return codes are the standard way to notify the caller of a problem in languages that do not have exceptions
- Example in C:

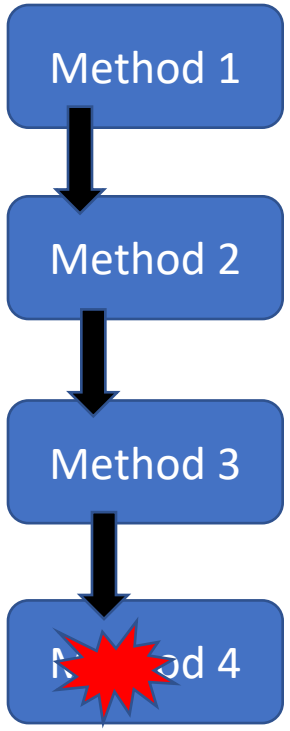
```
int file = open("data.txt", O_RDONLY);
if (file < 0) { /* could not open file */
  …
```

Challenges with Return Codes



- Advantages
  - Portable across languages
  - Default behaviour is to do nothing with the error code
    - If the error code is not checked, the program runs on
    - This is an advantage?
- Disadvantages
  - Default behaviour is to do nothing with the error code
  - Amount of information encoded in the return code is limited
  - Error handling code and normal code are mixed together
  - The error must be handled immediately by the caller
    - It is not automatically propagated to where it is best handled
- Most languages support a better system of error reporting: Exceptions

Exceptions: Throw, Propagate, Catch



1. An exception is **thrown** when an error occurs
   - The code identifies that an error has occurred
   - The code reports the error by throwing an exception
2. The exception is **propagated** up the call-chain until it is **caught** by a method that can deal with the error
   - The call-chain is the sequence of calls that have occurred to get to the current method
   - The propagation happens in reverse order of the method calls and continues until the exception is caught
3. The piece of code that **catches** the exception deals with the error
   - If the exception is never caught, the program is terminated, because the error is never dealt with.
   - Example of exceptions that are typically not caught:

     `NullPointerException`

Why Use Exceptions

- Each type of error is represented with a separate exception class
  - In Java, exceptions are objects that are instantiated from subclasses of the *Exception* class
- Exception objects store additional information about the error
  - When instantiating an exception object, the constructor is passed additional information
  - This information can later be used when the exception is caught
- The error handling code is separated from normal path code
  - Normal path code is placed in the **try** blocks
  - Exception handlers (catchers) are placed in the **catch** blocks
- **Note**: Exceptions are language dependent
  - Requires compiler support
  - Most modern languages have exceptions

Exception Handling Syntax in Java

- A **protected block** comprises 3 parts:
  - **`try`** : the common path code to be executed
  - **`catch`** : exception handlers for each exception to be caught
  - **`finally`** : an optional ”clean-up” handler that always runs after the ”try” regardless of whether an exception occurs
- Exceptions are **thrown** (or raised) by a `throw` statement

  `throw e;`

  where e is an Exception object, e.g.,

  `throw new Exception_1();`

```
try {
   // common path
} catch (Exception_1 e) {
   // Exception 1 handler
} catch (Exception_2 e) {
   // Exception 2 handler
} ...
} finally { // optional
   // clean up code
}
```

Example of Exception Use

```
Scanner scanner = new Scanner(System.in);

int mailboxId;
while (true) {
  try {
    System.out.print("Enter mailbox #:");
    mailboxId = scanner.nextInt();
    if ((mailboxId < 0) || (mailboxId > list.size())) {
      throw new IndexOutOfBoundsException();
    }
  } catch (InputMismatchException e) {
    System.out.println("Not an integer, re-enter");
  } catch (IndexOutOfBoundsException e) {
    System.out.println("Value out of range, re-enter");
  }
  break;
}

Mailbox mailbox = list.get(mailboxId);
...
```

Try block containing protected code

IndexOutOfBoundsException is thrown here

InputMismatchException is caught here

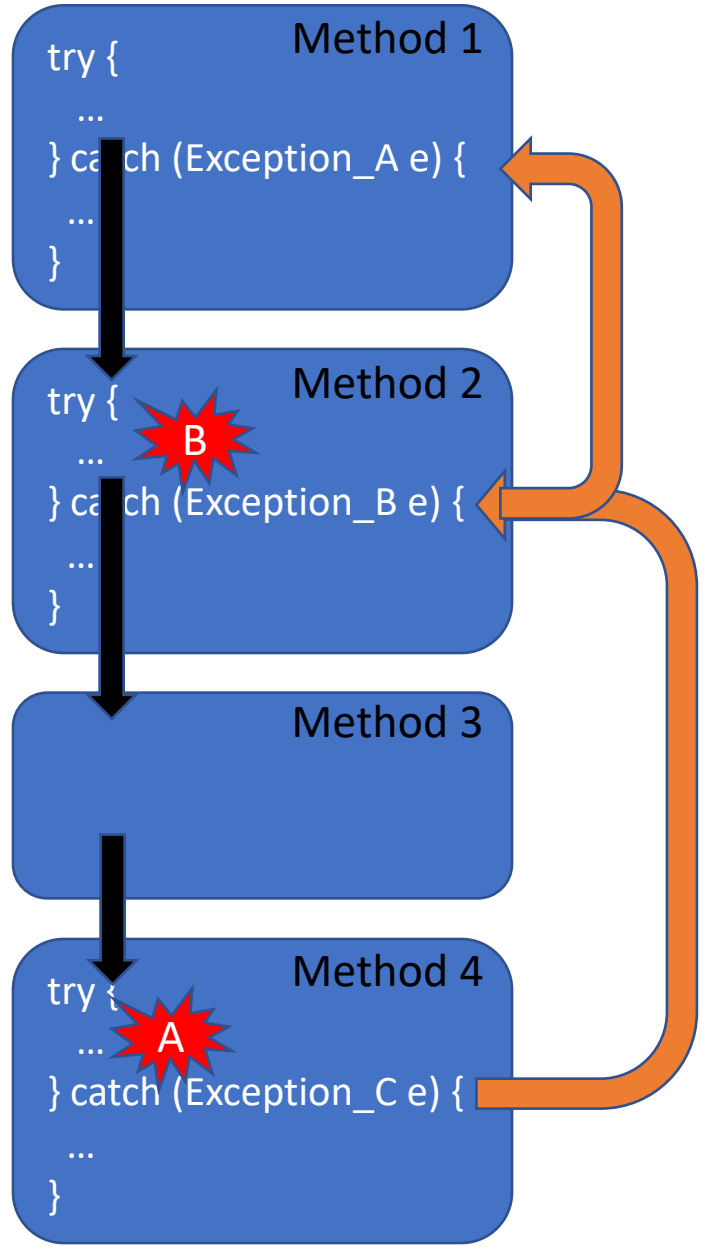
IndexOutOfBoundsException is caught here

If the user does not enter an integer, throw InputMismatchException

If value is not in range throw exception

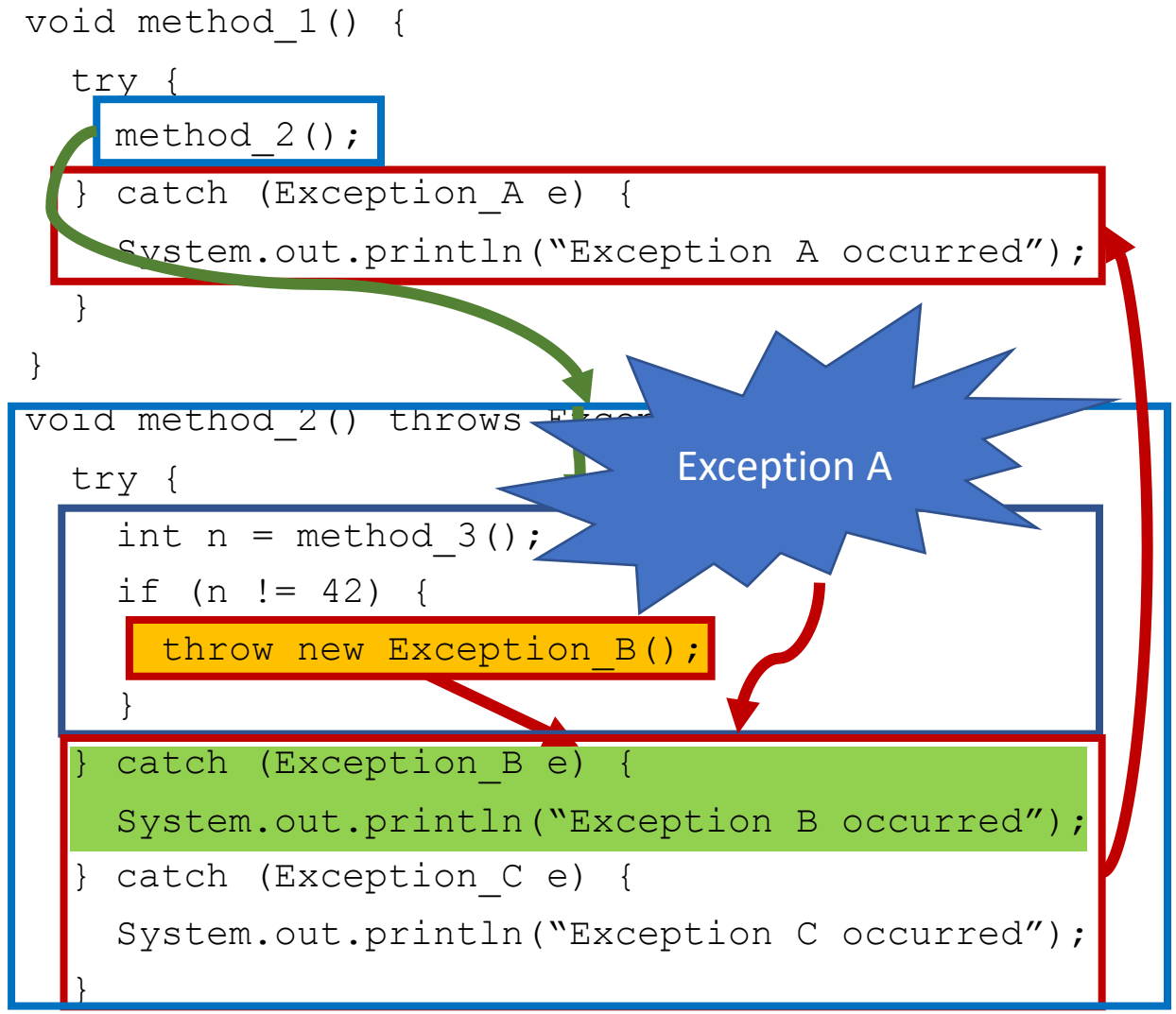
Exception Handling In Action

- Exception handlers are associated with a **try** block
- When an exception is thrown in the try block, search for a handler
- If there is no matching handler,
  - The execution jumps to the try block from which the current code was called
  - If there is no matching exception handler the call continues up the chain.
  - If the exception is never caught, the program crashes.



Exception Handling In Action

- Exception handlers are associated with a **try** block
- When an exception is thrown in the try block, search for a handler
- If there is no matching handler,
  - The execution jumps to the try block from which the current code was called
  - If there is no matching exception handler the call continues up the chain.
  - If the exception is never caught, the program crashes.



Methods that Throw Exceptions

- A method must explicitly declare that it throws an exception if
  - If code within the method throws an exception that is not caught
  - If code within the method does not catch an exception that is thrown by code that it calls
- For example: **`void method1() throws ExceptionA { … }`**
- It is a compile-time error in **Java** if a method does not declare its exceptions
  - In other languages, it is not always enforced by the compiler and must instead be **clearly documented.**

In Java, Exceptions are Objects

- Exceptions in Java all come from the same base class: *Exception*
- Every exception is instantiated from a class of the same name
- We create a new exception by subclassing the Exception class:
  - Example:

```
class MyException extends Exception {
  //Class name should indicate what caused the exception
}
```

- To throw an exception, we instantiate an object of that exception type
  - Example: `throw new MyException();`

Key 

Points

- Defensive programming focuses ensuring that our programs can recover and continue to run in the presence of bad data
- Defensive programming requires mechanisms that can
  - Report the error
  - Recover from the error
- Assertions are used to encode the assumptions in the code. Assertions report errors, but abort on error.
- Return codes are used to report errors but have many disadvantages
- Exceptions are a general mechanism for reporting errors

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://i.pinimg.com/474x/1a/64/88/1a64886a1f4c9176dae6b76144c577b4--roman-empire-public-domain.jpg
- https://clipartart.com/images/clipart-recovery-2.gif
- https://lh3.googleusercontent.com/proxy/vm-muRDjs_DUAHkcCbzcv4x0BYwz_TwHsLBln03mUQRnbE8vnaSqub92NI5_0k289CYJOplH1Tc4B2rek7ogVCZccebcAgUs_eqU6yrnYfg_U1XKYZLI-RmnuVn1
- https://cdn4.vectorstock.com/i/1000x1000/12/73/a-classic-cartoon-style-black-round-bomb-lit-on-vector-20631273.jpg