

Defensive Programming

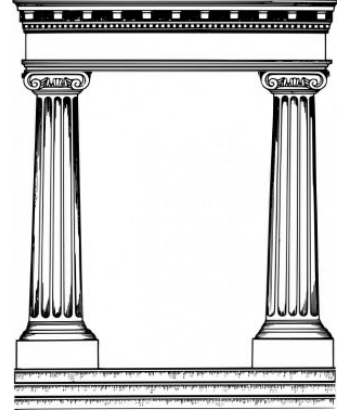
Part 2

CSCI 2134: Software Development

Agenda

- Lecture Contents
 - Best practices for Exceptions
 - Logging
 - Barricades
- Brightspace Quiz
- Readings:
 - This Lecture: Chapter 8
 - Next Lecture: Chapter 24

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

Error Handling

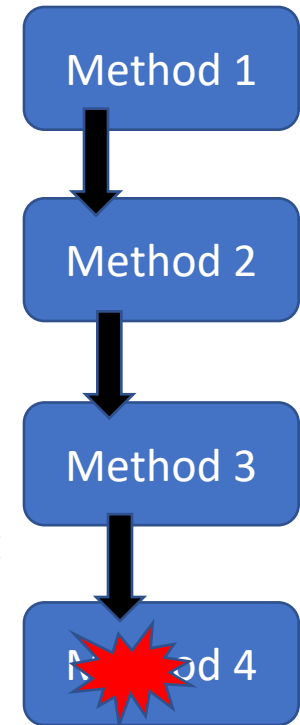
- Once we know there is a problem, we need to handle it
- An error occurs when the current computation cannot proceed due to bad data
- Two things occur when an error occurs:
 - **Recovery:** The program moves to a safe state from which it can proceed
 - **Report:** The program reports that an error 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 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



© Can Stock Photo

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 often not caught (but should be):
`NullPointerException`



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

Try block containing protected code

IndexOutOfBoundsException is thrown here

InputMismatchException is caught here

IndexOutOfBoundsException is caught here

```
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();
        }
        break;
    } catch (InputMismatchException e) {
        System.out.println("Not an integer, re-enter");
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Value out of range, re-enter");
    }
}

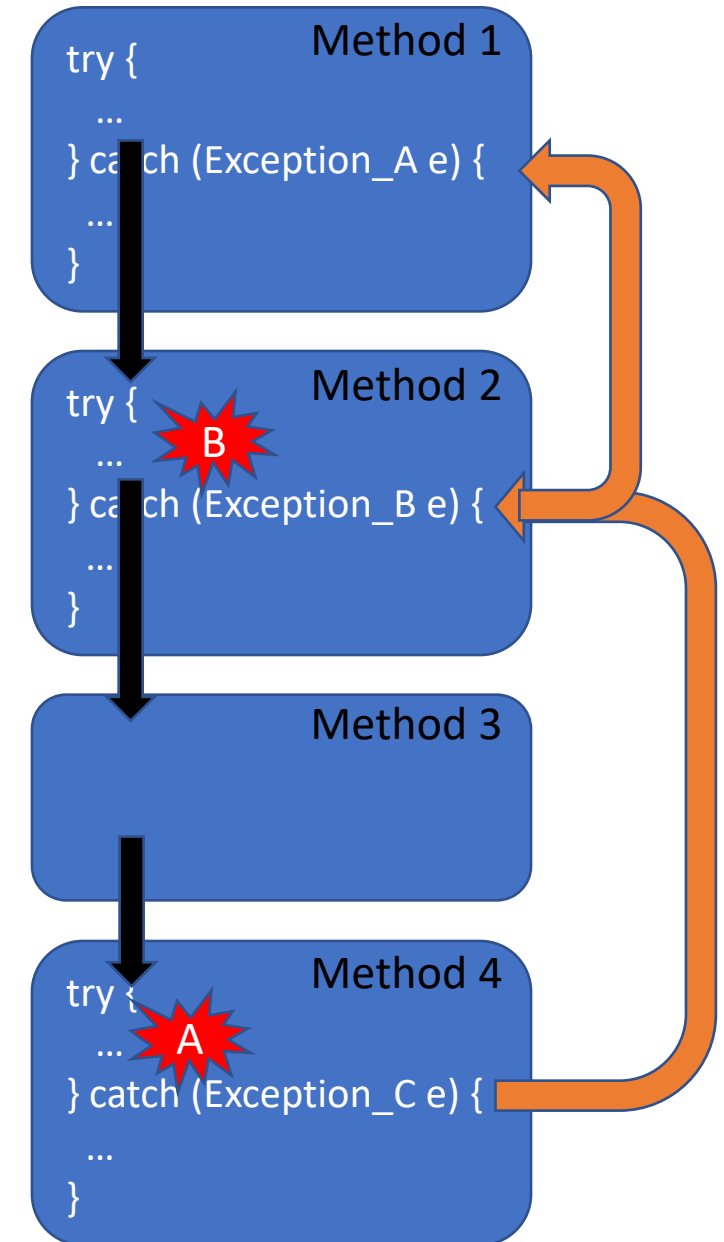
Mailbox mailbox = list.get(mailboxId);
...
```

If the user does not enter an integer, throw InputMismatchException

If value is not in range throw exception

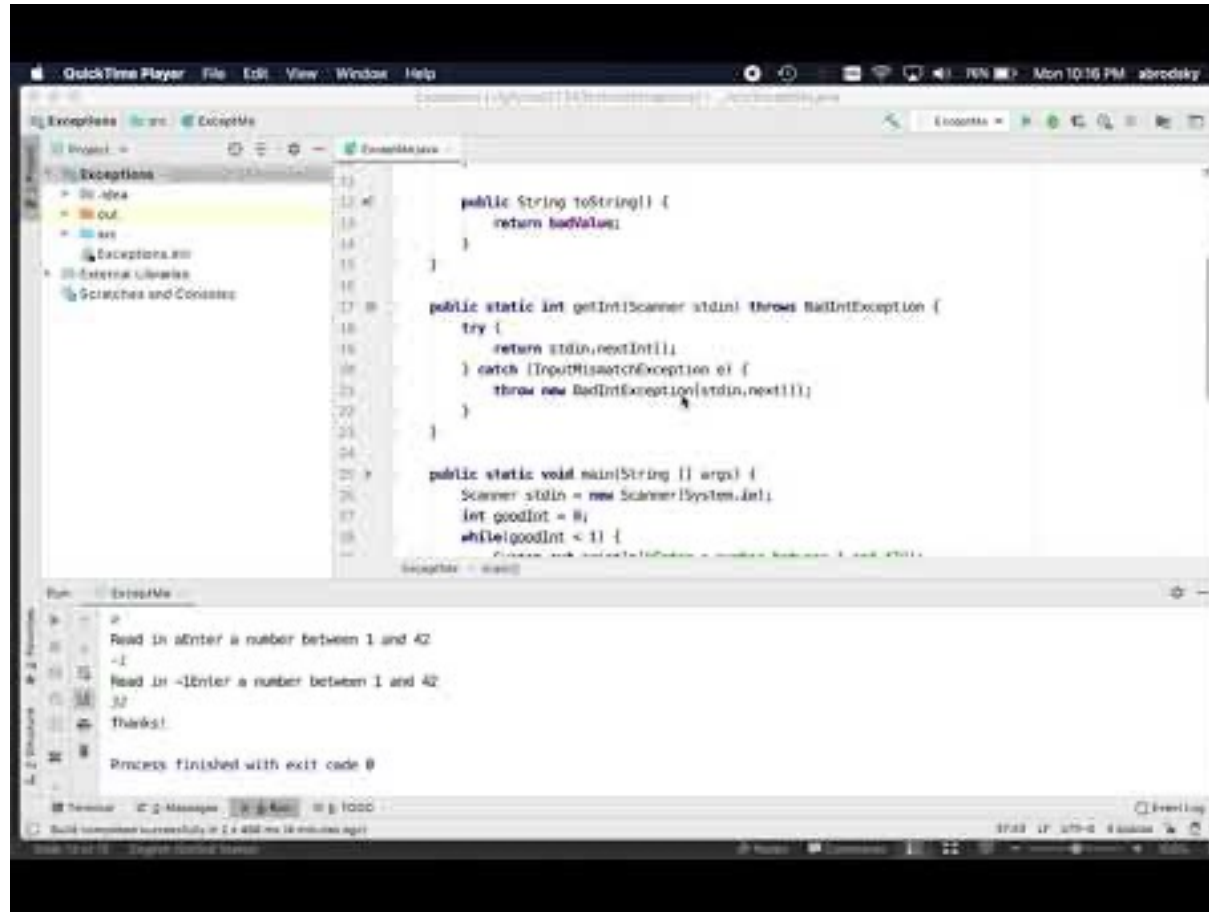
Exception Handling In Action

- 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.



Live Exceptions Demo

Follow along code: <https://git.cs.dal.ca/courses/2021-fall/csci-2134/lecture/exceptions.git>



The screenshot shows an IDE window titled 'QuickTime Player' with a menu bar (File, Edit, View, Window, Help). The main editor displays a Java file named 'Exceptions.java'. The code defines a 'BadIntException' class, a 'getIntScanner' method that throws 'BadIntException', and a 'main' method. The 'main' method uses a 'Scanner' to read input and checks if it's a valid integer between 1 and 42. If not, it throws a 'BadIntException'. The output window at the bottom shows the program's execution: it prompts for a number, reads '-1', prompts again, reads '32', and then prints 'Thanks!' and 'Process finished with exit code 0'.

```
11 public class BadIntException extends Exception {
12     public BadIntException(String message) {
13         super(message);
14     }
15 }
16
17 public static int getIntScanner(Scanner stdin) throws BadIntException {
18     try {
19         return stdin.nextInt();
20     } catch (InputMismatchException e) {
21         throw new BadIntException(stdin.next());
22     }
23 }
24
25 public static void main(String[] args) {
26     Scanner stdin = new Scanner(System.in);
27     int goodInt = 0;
28     while(goodInt < 1) {
29         System.out.println("Enter a number between 1 and 42:");
30         goodInt = getIntScanner(stdin);
31     }
32     System.out.println("Thanks!");
33 }
```

Run: Exceptions

Read in aEnter a number between 1 and 42
-1
Read in -1Enter a number between 1 and 42
32
Thanks!
Process finished with exit code 0

<https://youtu.be/3yipSsTynQs>

Example 1: Re-Allocate this!

- The code:

```
int n = scanner.nextInt();  
...  
int [][][] data =  
    new int[n][n][n];  
...
```

- What is the assumption?

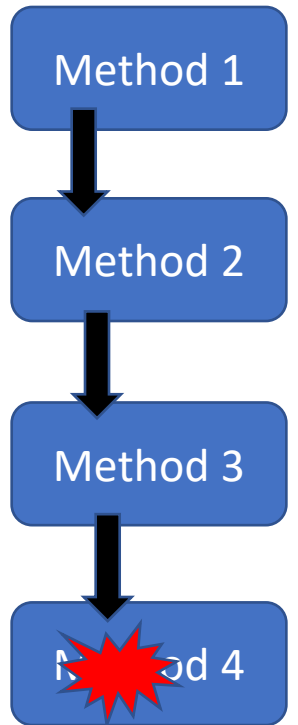
Programmer assumes the program can allocate sufficient memory

- The solution with exceptions:

```
int n = scanner.nextInt();  
...  
int [][][] data = null;  
  
try {  
    data = new int[n][n][n];  
} catch (OutOfMemoryError e) {  
    ...  
}  
...
```

Best Practice: Throw Early, Catch Late

- An exception should be thrown as soon as an error is detected
 - Do not throw an exception at the end of a method
 - Do not set an error flag and throw the exception later
 - Throw the exception immediately
- The exception should be caught by code that can deal with the exception
 - It's ok for exceptions to propagate up several levels



```
if (!isValid(data)) {  
    errorFlag = true;  
}  
...  
if (errorFlag) {  
    throw new OopsException();  
}
```

Bad

```
if (!(isValidData())) {  
    throw new OopsException();  
}  
...
```

Good

Best Practices:

Avoid using Exceptions for Normal Execution

- Exceptions are intended for exceptional conditions or error conditions
- They are not intended to replace the **If** statement

```
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();
        }
        break;
    } catch (InputMismatchException e) {
        System.out.println("Not an integer, re-enter");
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Value out of range, re-enter");
    }
}
```

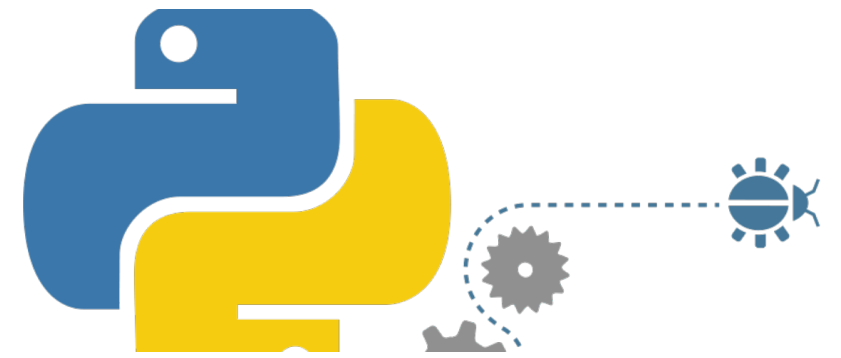
OK

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

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

Better

But... What about Python?



- Aside: In the Python community, it is accepted that exceptions can be used to deal with issues that would normally be part of the normal control flow?
- **Question:** Why is it best practice not to use exceptions for normal control flow?

Exceptions are computationally expensive. They can have a significant performance impact
- **Question:** Why is Python different?

Python is an interpreted language, where performance is much less of a consideration.
Hence, the use of exceptions for normal control flow is accepted.

Best Practice:

Never Swallow Exceptions (Don't Mask Bugs!)

- Runtime exceptions (errors) can be caught
 - `NullPointerException`
 - `OutOfMemoryError`
 - `IndexOutOfBoundsException`
 - etc
- These are symptoms of bugs that should be fixed!
- **Do not ignore the problem!**

```
try {  
    // Do stuff that causes  
    // an error  
} catch (Throwable e) {  
    // Do nothing 😞  
}  
...
```

Best Practice:

Do not Create New Exceptions Unless Necessary

- Why Not?
 - There are already many standard exceptions to cover most situations
 - Introducing new exceptions increases complexity
 - Remember:
 - Code is a liability.
 - Less is better.
- When Is it Ok to create new exceptions?
 - When existing exceptions do not describe the error condition
 - When existing exceptions cannot store the information that needs to be propagated

Best Practice: Set a Global Exception Handler

- Set a global exception handler to
 - Catch all exceptions not caught by your code
 - Inform the user of the error
 - Log the error
 - Notify developers
 - Etc
- Note: This is a language supported feature

Example: Set a Global Exception Handler

(<https://www.nomachetejuggling.com/2006/06/13/java-5-global-exception-handling/>, 12/02/2020)

```
public class ExceptionHandlerExample {  
    public static void main(String[] args) throws Exception {  
  
        Handler handler = new Handler();  
        Thread.setDefaultUncaughtExceptionHandler(handler);  
  
        // Rest of main method  
    }  
}
```

Instantiate a handler object

Set the handler object as the default handler

```
class Handler implements Thread.UncaughtExceptionHandler {  
    public void uncaughtException(Thread t, Throwable e) {  
        System.out.println("Throwable: " + e.getMessage());  
        System.out.println(t.toString());  
    }  
}
```

The code in the Handler class should log the error and notify the user, etc.

Other Best Practices

- Use exception in error situations that you anticipate but the cause is out of your control
- Be aware which exceptions your code must catch
 - Catch **specific** exceptions
 - E.g., catch `InputMismatchException` rather than `Exception`
 - Why? Self-documenting. The more specific, the easier it is to deal with!
- Exceptions should **include enough information to understand the error**
 - What good is an error happening if we don't collect information on how to avoid it in the future?
 - Logging

Logging



- **Problem:** Error reports need to be recorded
 - In many cases the developer is not the user of the software
 - Software will manifest bugs when it is used by the user
 - Need some way to record what happened so that the issue can be debugged
- **Idea:** Use a log to record the events in your program
 - A log is a chronological record of all “important” events
 - The log can be reviewed for anomalies or behaviours
- Examples of events:
 - Accesses and log-ons
 - Software errors
 - Actions taken by the software



Log Storage

The log can be

- A file
 - Most common
- A database
- A console/terminal
 - Not very permanent
- A network location
- Anywhere that can store and retrieve the log records

```
Tue Feb 11 10:19:30 2020 thor[5006]: Connection from 134.190.176.229:55606
Tue Feb 11 10:19:35 2020 thor[5006]: Connection closed to 134.190.176.229:55606
Tue Feb 11 10:19:50 2020 thor[5007]: Connection from 134.190.176.229:55614
Tue Feb 11 10:22:23 2020 thor[5008]: Connection from 134.190.238.146:51309
Tue Feb 11 10:22:44 2020 thor[4957]: read() failed: Connection timed out
Tue Feb 11 10:22:44 2020 thor[4957]: Connection closed to 134.190.161.209:50204
Tue Feb 11 10:25:06 2020 thor[5008]: Connection closed to 134.190.238.146:51309
Tue Feb 11 10:35:52 2020 thor[5007]: read() failed: Connection reset by peer
Tue Feb 11 10:35:52 2020 thor[5007]: Connection closed to 134.190.176.229:55614
Tue Feb 11 10:42:24 2020 thor[5031]: Connection from 134.190.172.202:55932
Tue Feb 11 10:45:15 2020 thor[4999]: Connection closed to 134.190.148.199:57265
Tue Feb 11 10:45:30 2020 thor[5036]: Connection from 134.190.148.199:53883
Tue Feb 11 10:53:43 2020 thor[5036]: Connection closed to 134.190.148.199:53883
Tue Feb 11 10:53:50 2020 thor[5049]: Connection from 134.190.148.199:53957
Tue Feb 11 10:58:59 2020 thor[5049]: Connection closed to 134.190.148.199:53957
Tue Feb 11 11:14:44 2020 thor[5031]: Connection closed to 134.190.172.202:55932
Tue Feb 11 11:31:50 2020 thor[5097]: Connection from 134.190.173.99:64044
Tue Feb 11 11:44:38 2020 thor[5114]: Connection from 134.190.148.199:54433
Tue Feb 11 11:45:28 2020 thor[5114]: Connection closed to 134.190.148.199:54433
Tue Feb 11 11:47:57 2020 thor[5097]: Connection closed to 134.190.173.99:64044
Tue Feb 11 11:57:54 2020 thor[5127]: Connection from 134.190.161.209:50550
Tue Feb 11 11:58:03 2020 thor[5127]: Connection closed to 134.190.161.209:50550
```

Types of Logs

Informational Log

- **Purpose:** To be reviewed by humans (users or developers)
- **Format:** Typically text based
- **Information:**
 - Data / Time
 - Process
 - Event type
 - Related information



Replay Log

- **Purpose:** To be rerun on the software to simulate a user session
- **Format:** Typically binary
- **Information:**
 - Initial system state
 - Input to system
 - Time of each input



Informational Log Entries

- A log is a chronological series of entries (or records)
- Each entry should contain
 - Date / Time of event [required]
 - Process name [recommended]
 - Allows the same log file to be used for several programs
 - Severity level [recommended]
 - Description of event [required]
 - Information associated with the event [recommended]

Date / Time

Program

Tue Feb 11 10:19:30 2020 thor[5006]: Connection from 134.190.176.229:55606
Tue Feb 11 10:19:35 2020 thor[5006]: Connection closed to 134.190.176.229:55606
Tue Feb 11 10:19:50 2020 thor[5007]: Connection from 134.190.176.229:55614
Tue Feb 11 10:22:23 2020 thor[5008]: Connection from 134.190.238.146:51309
Tue Feb 11 10:22:44 2020 thor[4957]: read() failed: Connection timed out
Tue Feb 11 10:22:44 2020 thor[4957]: Connection closed to 134.190.161.209:50204
Tue Feb 11 10:25:06 2020 thor[5008]: Connection closed to 134.190.238.146:51309
Tue Feb 11 10:35:52 2020 thor[5007]: read() failed: Connection reset by peer
Tue Feb 11 10:35:52 2020 thor[5007]: Connection closed to 134.190.176.229:55614
Tue Feb 11 10:42:24 2020 thor[5031]: Connection from 134.190.172.202:55932
Tue Feb 11 10:45:15 2020 thor[4999]: Connection closed to 134.190.148.199:57265
Tue Feb 11 10:45:30 2020 thor[5036]: Connection from 134.190.148.199:53883
Tue Feb 11 10:53:43 2020 thor[5036]: Connection closed to 134.190.148.199:53883
Tue Feb 11 10:53:50 2020 thor[5049]: Connection from 134.190.148.199:53957
Tue Feb 11 10:58:59 2020 thor[5049]: Connection closed to 134.190.148.199:53957
Tue Feb 11 11:14:44 2020 thor[5031]: Connection closed to 134.190.172.202:55932
Tue Feb 11 11:31:50 2020 thor[5097]: Connection from 134.190.173.99:64044
Tue Feb 11 11:44:38 2020 thor[5114]: Connection from 134.190.148.199:54433
Tue Feb 11 11:45:28 2020 thor[5114]: Connection closed to 134.190.148.199:54433
Tue Feb 11 11:47:57 2020 thor[5097]: Connection closed to 134.190.173.99:64044
Tue Feb 11 11:57:54 2020 thor[5127]: Connection from 134.190.161.209:50550
Tue Feb 11 11:58:03 2020 thor[5127]: Connection closed to 134.190.161.209:50550

Event description

Event information

Determining What to Log

- Many logging systems provide different levels of severity
 - Error or Severe
 - Warning
 - Information
 - Configuration
 - Etc ...
- **Best Practice:** Control what is logged at the logging facility
 - Add code to log all possible events of interest
 - Assign a severity level to each event
 - Configure the logging facility to log all events at a minimum severity
E.g., Information or higher...

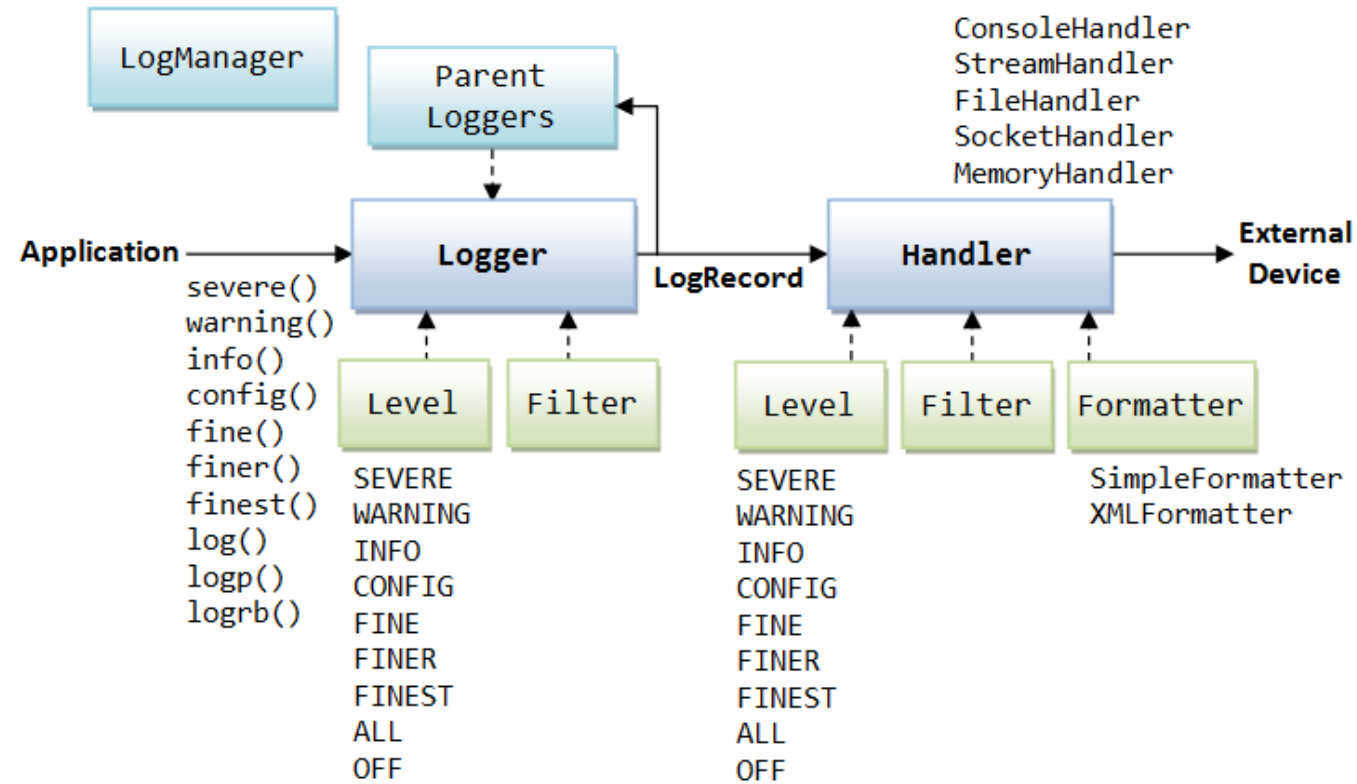
Logging Best Practices

1. Create log messages / information
 - In a readable format
 - Know your audience
 - In a machine-parseable format: XML, JSON, Key-value pairs, CSV, SQL
2. Include levels or types for your log messages
3. Log more than just debugging events
 - Too much is better than not enough
4. Identify the location of the event
 - Include context and additional information
5. Ensure that your logs can be easily and centrally accessed
6. Don't write your own logging system if you can avoid it!

Java Logging

Never write what you don't have to

- Do not write your own logging system!!
- Use something that exists like **java.util.logging**
 - Provides more consistency
 - Less code for you to maintain



Example: Global Exception Handler w/ Logging

(<https://www.nomachetejuggling.com/2006/06/13/java-5-global-exception-handling/>, 12/02/2020)

```
public class ExceptionHandlerExample {  
    public static void main(String[] args) throws Exception {  
        Logger log = new MainLogger("Project X", bundle);  
  
        Handler handler = new Handler(log);  
        Thread.setDefaultUncaughtExceptionHandler(handler);  
  
        // Rest of main method  
    }  
}
```

Create a log

Instantiate a
handler object
with the log

Set the handler
object as the
default handler

```
class Handler implements Thread.UncaughtExceptionHandler {  
    Logger log;
```

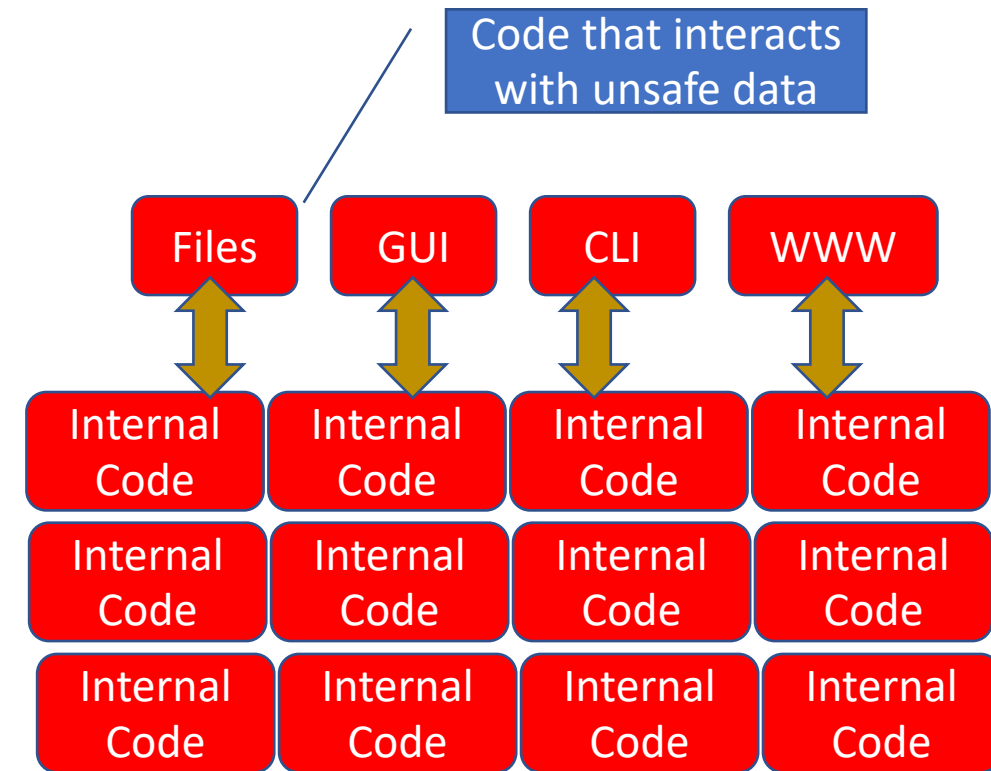
Store a ref to
the log in
handler

```
    public Handler(Logger log) { this.log = log; }  
  
    public void uncaughtException(Thread t, Throwable e) {  
        log.severe("Throwable: " + e.getMessage() + "\n" + t.toString());  
    }  
}
```

Log an error

Motivation for Barricades

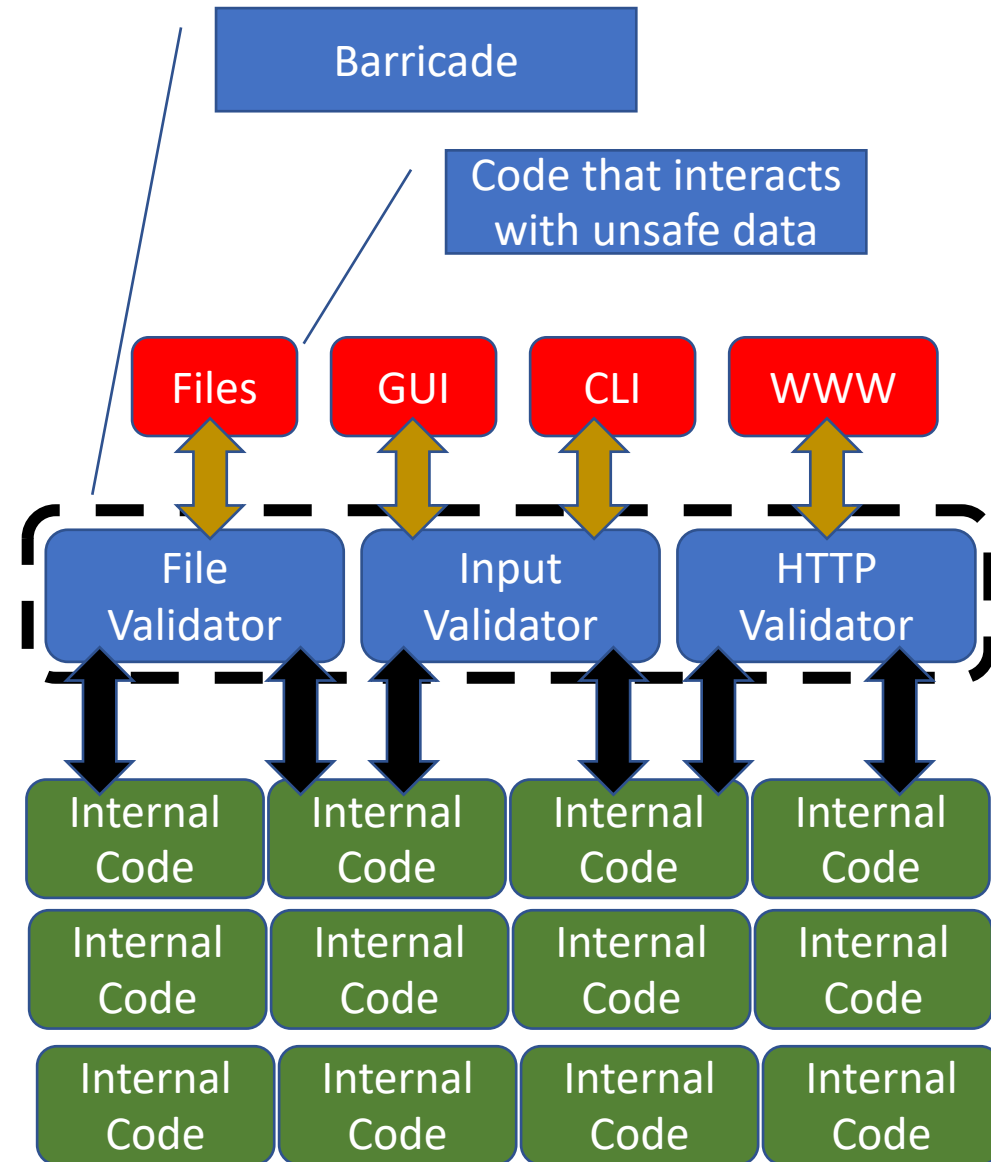
- **Problem:** Defensive programming has a cost
 - Validating all data everywhere is expensive
 - Adds more code
 - Creates more work
- We would like a security blanket (or barricade) around part of our code to make it simpler



Barricades: A Compromise

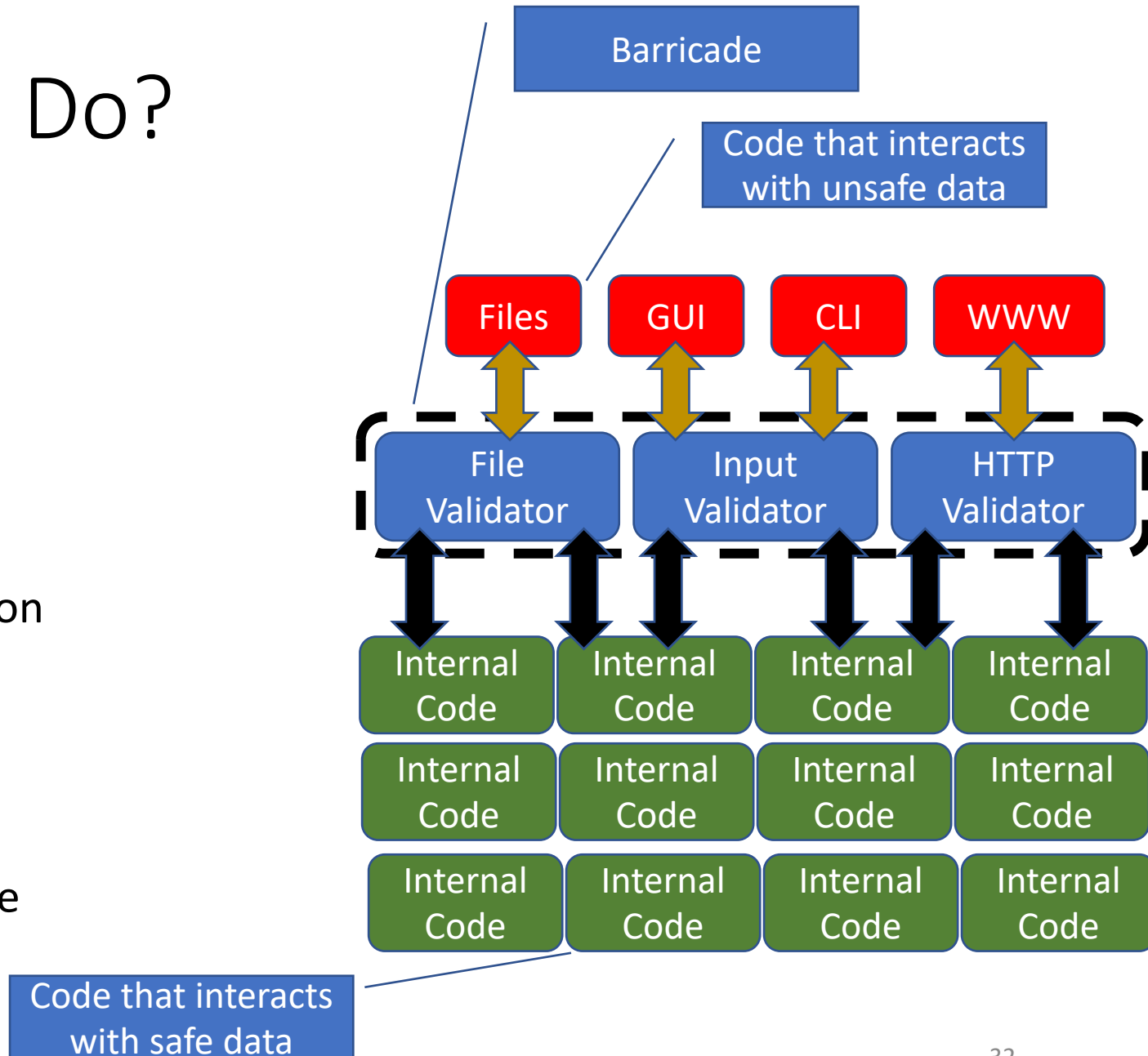
- **Problem:** Defensive programming has a cost
 - Validating all data everywhere is expensive
 - Adds more code
 - Creates more work
- **Idea:** Only do strong validation when data moves from one module to another
 - All methods part of the public API for a module perform strong data validation
 - All private and protected methods, which are used internally by a module assume that the data has been validated

Code that interacts with safe data



What do Barricades Do?

- Validate data: Check for
 - Correct format
 - Illegal characters
 - Correct ranges
 - Correct enumeration
 - Consistency
 - Etc.
- Convert data to internal representation
 - E.g.
 - From text to binary
 - Text sequences to lists
 - Relationships to graphs
 - Etc.
- Pass the safe data to the interval code
- Example: JSON parser



When to use Barricades?

- When there are natural boundaries between parts of the code
E.g., User interface vs business logic
- When there are multiple interacting systems

Key Points

- Exceptions are a general mechanism for recovering from and reporting errors in the code
- Exceptions are thrown when an error occurs and are caught by part of the code that can deal with an error
- Logging is used to create a record of events and errors that took place during an execution of a piece of code.
- Logging is useful for debugging and general auditing of systems
- Barricades are a strategy to separate unsafe code (code that works with unvalidated data) from safe code (code that works with validated data) and ensure that only validated data is passed to the safe code.

Image References

Retrieved January 29, 2020

- <http://pengetouristboard.co.uk/vote-best-takeaway-se20/>
- https://cdn-images-1.medium.com/fit/t/1600/480/1*WWrXceae4H_klzpPU6h7Hg.png
- <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_TwHsLBIn03mUQRnbE8vnaSqub92NI5_0k289CYJOplH1Tc4B2rek7ogVCZccebAgUs_eqU6yrnYfg_U1XKYZLI-RmnuVn1
- https://lh3.googleusercontent.com/proxy/0WbT8CAd34kox5aVkBqZUSjgZhZxs2E7JAukV-d-t7vc2_M_bsy-UhgsxG3kVQ1lgnf1EP8cuq33T2q7VUQNH3H
- https://lh3.googleusercontent.com/proxy/eUjHn4-xChdnHDahGAjmWmcSDQyDtqHE6ecS8KK3FU7vhH7AFJPIBX8Gn-iWaLwApI08lpVJMrFg4-Rdoxij14yRilbGX_D4h82IIb0dpKzfWbamW9pKaJDm2uKf4
- https://www.pincliptart.com/picdir/middle/110-1108784_what-is-information-poverty-information-png-clipart.png
- <https://i1.pngguru.com/preview/950/521/306/icon-relieve-azul-replay-png-clipart-thumbnail.jpg>
- https://pics.clipartpng.com/Yellow_Road_Barricade_PNG_Clip_Art-1345.png

Retrieved October 16, 2020

- <https://www.oracle.com/technical-resources/articles/middleware/soa-luttikhuizen-fault-handling-1.html>