Defensive Programming

Dr. Isaac Griffith

Idaho State University

Outcomes

After today's lecture you will be able to:

- Understand the basic ideas of defensive coding
- Understand how exceptions relate to defensive coding and fault tolerant coding

Defensive Programming

CS 2263

ROAR

Defensive Programming

- Programming defensively means making your code **robust** to unexpected use.
- Use the **need to know** principle: Only expose the parts of your class that your client classes need to know
- Java exceptions provide a uniform way of handling errors

Why Program Defensively?

- Normally, your classes will form part of a larger system
- So other programmers will be using and relying upon your classes
- Obviously, your classes should be *correct*, but equally importantly, you classes should be *robust* – that is, resistant to accidental misuse by other programmers
- You should aim to ensure that no errors in the final system can be attributed to the behavior of your classes
- We use the terminology *“client code”* for the code written by other programmers that are using your classes.

Error Handling

CS 2263

Error Handling Concepts

- Murphy's Law
  - "Anything that can go wrong will go wrong"
- Error conditions will occur, and your code needs to deal with them
  - Out of memory, disk full, file missing, file corrupted, network error, …
- Software should be tested to see how it performs under various error conditions
  - Simulate errors and see what happens
- Just because your program works on your computer doesn't mean that it will work everywhere else
- You'll be amazed at how many weird things will go wrong when your software is used out in the "wild"

Error Handling Concepts

- What should a program do when an error occurs?
- Some errors are “recoverable” - the program is able to recover and continue normal operation
- Many errors are “unrecoverable” - the program cannot continue and gracefully terminates
- Most errors are detected by low-level routines that are deeply nested in the call stack
- Low-level routines usually can’t determine how the program should respond
- Information about the error must be passed up the call stack to higher-level routines that can determine the appropriate response

Propagating Error Information

- Return Codes
- Status Parameter
- Error State
- Exceptions

Exceptions

CS 2263

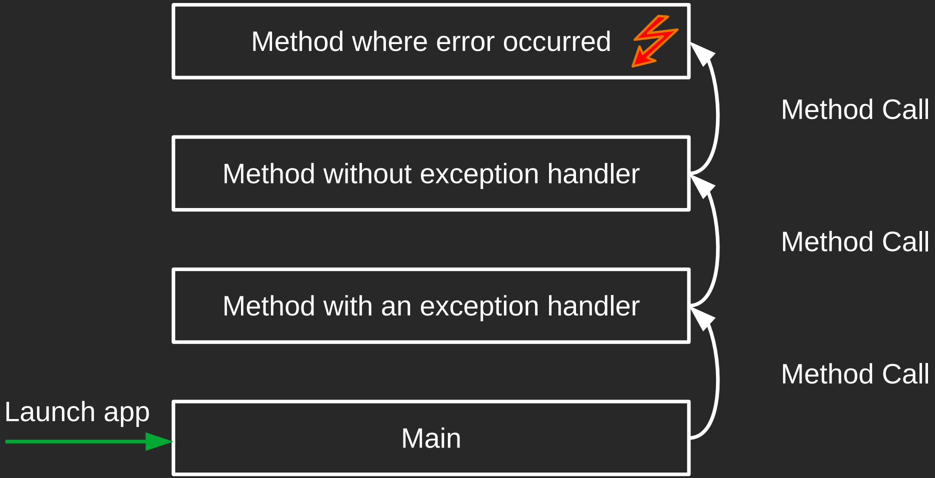
Why Exceptions

- Exceptions are an elegant mechanism for handling errors without the disadvantages of the other techniques
  - Return values aren't tied up
  - No extra parameters
  - Error handling code isn't mixed in with the "normal" code
  - You can't ignore exceptions - if you don't handle them, your program will crash

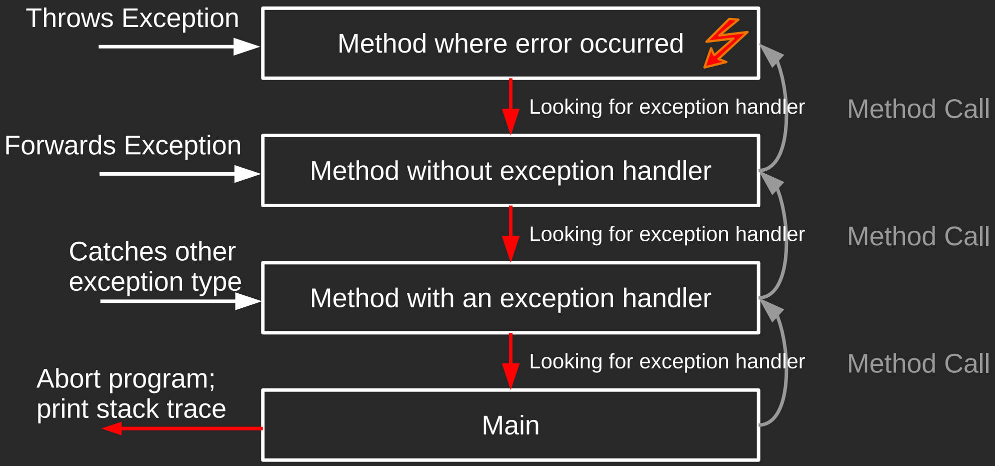
- After an exception is thrown, the runtime will try to locate the relevant **exception handler**

- Runtime **searches back** through the call stack and will stop at the first relevant exception handler

Tracing the Call Stack



Re-Tracing the Call Stack



Catch or Specify

- Requirement for code that **might throw exception**:
  - Possess a `try` statement to catch exception
  - Method specifies that the exception can be thrown using the `throws` clause

Step 1: Add Try Block


```java
private List<Integer> list;
private static final int SIZE = 10;

public void writeList() {
  PrintWriter out = null;
  try {
    // Exception thrown somewhere within this block
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
      out.println("Value at: " + i + " = " + list.get(i));
    }
  } // End of try block
  //... catch and finally blocks ...
}
```

Step 2: Add Catch Block

Idaho State University | Computer Science

```java
try {
  // Exception thrown somewhere within this block
  System.out.println("Entered try statement");
  out = new PrintWriter(new FileWriter("OutFile.txt"));
  for (int i = 0; i < SIZE; i++) {
    out.println("Value at: " + i + " = " + list.get(i));
  }
} catch (IndexOutOfBoundsExceptoin e) {
  System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
  System.err.println("Caught IOException: " + e.getMessage());
}
```

Step 3: Add Optional Finally Block

```
finally {
  if (out != null) {
    System.out.println("Closing PrintWriter");
    out.close();
  } else {
    System.out.println("PrintWriter not open");
  }
}
```

- Finally block is always executed
- Useful place to perform cleanup work after success or fail
- Typical usage is to release resources by calling **close()**
- Avoids resource leaks

Try-with-resource Alternative

```java
static String readFirstLineFromFile(String path) throws IOException {
  try (BuffererdReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

- Try statement that declares one or more resources
- Resources are objects that must be released after use
- Requires the object to implement `java.lang.Autoclosable`

ROAR

Using throws clause

- The current method may not always be the appropriate place to deal with an exception

- Instead, exception handling can be located elsewhere and exceptions forwarded up the call stack

```
public void writeList() throws IOException, IndexOutOfBoundsException {
  PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
  for (int i = 0; i <s SIZE; i++) {
    out.println("Value at: " + i + " = " + list.get(i));
  }
  out.close();
}
```

Do we need both to be declared?

Using throw statement

- Exceptions can be generated from any point in a program
- Simply `throw new ExceptionType;`

```java
public Object pop() {
  Object obj;

  if (size == 0) {
    throw new EmptyStackException();
  }

  obj = objectAt(size - 1);
  setObjectAt(size - 1, null);
  size--;
  return obj;
}
```

Best Practices

- Use exceptions only for exceptional conditions

```
// Horrible abuse of exceptions. Don't ever do this
try {
  int i = 0;
  while (true)
    range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

- Use checked expressions for recoverable conditions and runtime exceptions for programming errors
  - e.g. File not found vs. array indexing problem

Best Practices

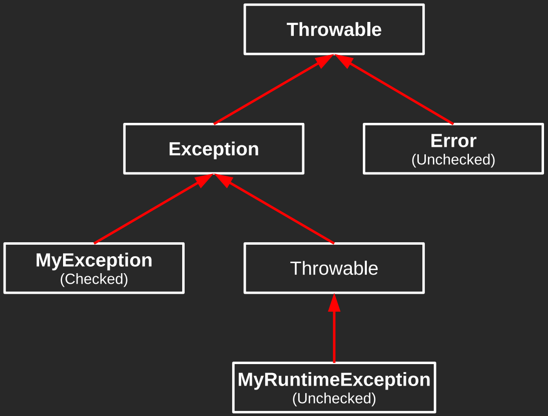
- Avoid unnecessary use of checked exceptions
  - Creates a difficult to use API
- Favor the standard exceptions:
  - `IllegalArgumentException`, `IllegalStateException`
  - `NullPointerException`, `IndexOutOfBoundsException`
  - `ConcurrentModificationException`
  - `UnsupportedOperationException`
- Document all exceptions thrown by methods
- Include failure-capture information in detailed messages
- Don't ignore exceptions

Kinds of Exceptions

- **Checked Exception**
  - Application should anticipate and recover from
  - e.g., `java.io.FileNotFoundException`
- **Error**
  - Circumstances external to the application
  - e.g., Hardware Failure
  - Cannot be caught
- **Runtime Exception**
  - Internal to the application, typically bugs
  - e.g., `NullPointerException` (can be caught, but better to abort and fix)
  - Do not need to be specified

For the Lazy Programmer…

- Both Error and RuntimeExceptions are **unchecked exceptions**
- Programmers can avoid the catch or specify requirement by **extending** their exception classes from `Error` or `RuntimeExceptions`
- Silences the compiler :-)



Assertions

CS 2263

Assertions

- As we write code, we make many assumptions about the state of the program and the data it processes
    - A variable's value is in a particular range
    - A file exists, is writable, is open, etc.
    - The maximum size of the data is N (e.g., 1000)
    - The data is sorted
    - A network connection to another machine was successfully opened
    - …
- The correctness of our program depends on the validity of our assumptions
- Faulty assumptions result in buggy, unreliable code

Assertions

Idaho State University | Computer Science

- Boolean expressions
- Used to check:
  - Pre-conditions
    - reflect **requires** clause
    - Test client
  - Post-conditions
    - reflect **effects** clause
    - test procedure
  - Invariants
- Include specification in the software

Invariants

- **Invariant** – "A rule, such as the ordering of an ordered list or heap, that applies throughout the life of a data structure or procedure. Each change to the data structure must maintain the correctness of the invariant"

- **Class Invariant** – if the "data structure" above is a class

Invariants Example

```
class CharStack {
  private char[] cArr; // internal rep
  private int i = 0;
  void push (char c) {
    cArr[i] = c;
    i++;
  }
}
```

- The invariant in this example is: "i should always be equal to the size of the stack (i.e., point at one above at the top of the stack)"

Assertions in Java

- Added in JDK 1.4

- General Syntax:

```
assert expression1 : expression2
```

- Examples:

```
assert value >= 0;
assert someInvariantTrue();
assert value >= 0 : "Value must be > 0: value = " + value;
```

- `> javac *.java`

- `> java -ea MyClass`

Handling Assertions in Java

- Evaluate $expression_1$
  - If true
    - No further action
  - If false
    - And if $expression_2$ exists Evaluate $expression_2$ and throw AssertionError(expression2)
  - Else
    - Use the default AssertionError constructor

Care with Assertions

- Side effects in assertions

```
void push (char c) {
  cArr[i] = c;
  assert (i++ == topElement());
}
```

- Change of flow in assertions

- Performance vs. correctness
  - Open issue

Assertions vs. Exceptions

- If one of my assumptions is wrong, shouldn't I throw an exception rather than use an assertion?
- Assertions are used to find and remove bugs before software is shipped
  - Assertions are turned off in the released software
- Exceptions are used to deal with errors that can occur even if the code is completely correct
  - Out of memory, disk full, file missing, file corrupted, network error, …

Assertions vs. Exceptions


```java
// In Class Sensor:
public void setSampleRate(int rate) throws SensorException {
    if (rate < MIN_HERTZ || MAX_HERTZ < rate)
      throw new SensorException("Illegal rate: " + rate);

    this.rate = rate;
}

public void setSampleRate(int rate) {
  assert MIN_HERTZ <= rate && rate <= MAX_HERTZ :
    "Illegal rate: " + rate;

  this.rate = rate;
}
```

Parameter Checking

- Another important defensive programming technique is “parameter checking”
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?

Parameter Checking

- Another important defensive programming technique is “parameter checking”
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?
- If you have control over the calling code, use asserts
  - If parameter is invalid, you can fix the calling code
- If you don’t have control over the calling code, throw exceptions
  - e.g., your product might be a class library

For Next Time

- Review this lecture
- Watch Lecture 25

Are there any questions?

ROAR