Exception Handling

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Anticipate problems beyond program control that may occur at runtime

Inspiration

“The best error message is the one that never shows up.” – Thomas Fuchs

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

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

Return Codes

- A method uses its return value to tell the caller whether or not it succeeded
- In case of failure, the particular value returned can be used to determine the nature of the error

```
int openFile(string fileName) {
  // ...
}

int result = openFile("index.html");
if (result < 0) {
  switch (result) {
    case -1: // file doesn't exist
    case -2: // file isn't writable
    case -3: // max number of files already open
  }
}
```

Return Codes

- Disadvantages of return codes
  - You have to use the return value to return error info even if you'd rather use it to return something else
  - Every time you call a method, you need to write code to check the return value for errors
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check return values

Status Parameter

- A method has an additional parameter through which it returns status information
- In case of failure, the particular value returned through the parameter can be used to determine the nature of the error

```
enum Status { GOOD, NOEXIST, NOWRITE, MAXOPEN };
void openFile(String filename, Status status) { ... }

Status status;
openFile("index.html", status)
if (status != Status.GOOD) {
  switch (status) {
    case NOEXIST: // file doesn't exist
    case NOWRITE: // file isn't writable
    case MAXOPEN: // max number of files already open
  }
```

Status Parameter

- Disadvantages of status parameters
  - Every method call has an extra parameter (but you can use the return value for whatever you want)
  - Every time you call a method, you need to write code to check the status parameter's value for errors
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check the status parameter

Error State

- Methods don't return error info
  - If something went wrong, you can't tell
- Objects store error info internally
- If you want to know if failures have occurred, you must query the object by calling a method

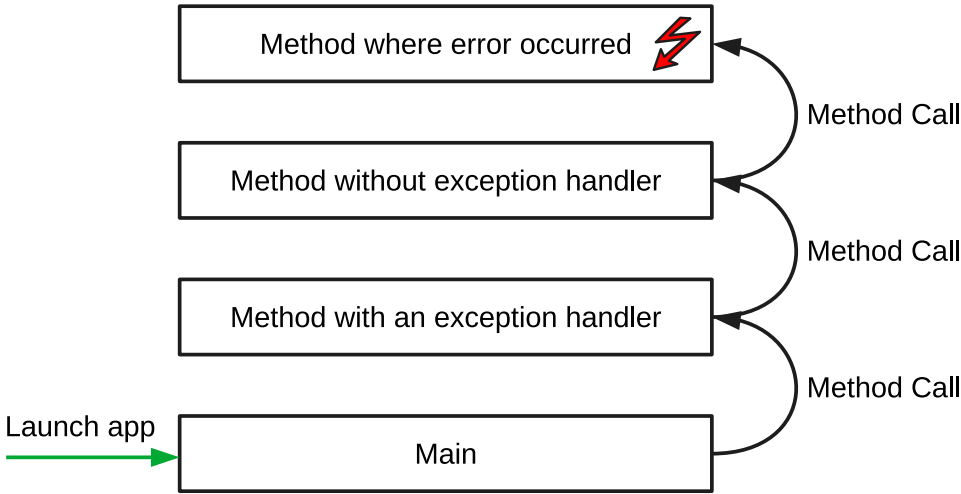
Error State

- Disadvantages of error state
  - Every time you call a method, you need to write code to check the object's error state
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check the error state

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

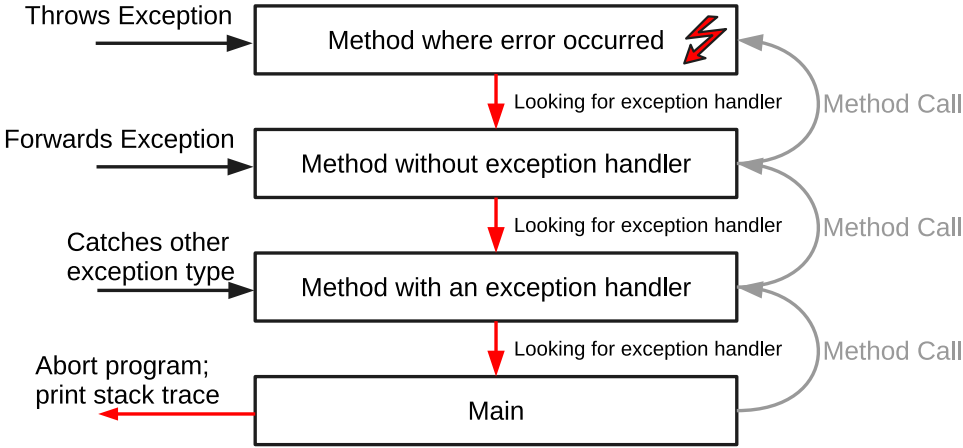
Tracing the Call Stack



Exception Handling

- 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

Re-Tracing the Call Stack



Catch or Specify

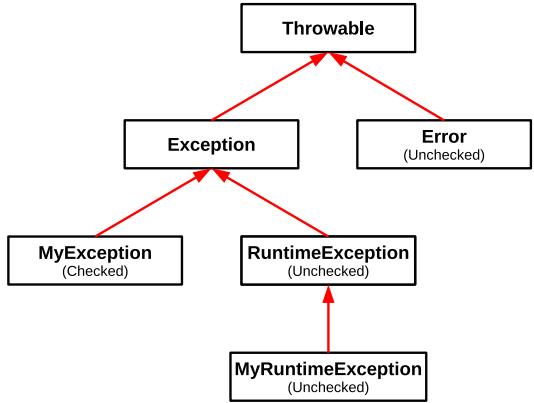
- Requirement for code that **might throw exception**:
  - Possess a `try` statement to catch exception
  - Method specifies that the exception can be thrown using the `throws` clause

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 :-)



Catching and Handling

- Exception handling components
  - `try`
  - `catch`
  - `finally`
  - `try-with-resource`

```java
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {
    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers() {
      list = new ArrayList<>(SIZE);
      for (int i = 0; i < SIZE; i++) {
        list.add(new Integer(i));
      }
    }

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

Locate two potential exceptions

Step 1: Add Try Block

```
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

```
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

Initial Form

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

Final Form

```
public void writeList() {
  PrintWriter out = null;
  try {
    // Exception throwm 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());
  } finally {
    if (out != null) {
      System.out.println("Closing PrintWriter");
      out.close();
    } else {
      System.out.println("PrintWriter not open");
    }
  }
}
```

Try-with-resource Alternative

```
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`

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 < 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;`

```
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

Checked vs. Unchecked

- Checked
  - compiler forces handling exceptions
  - must handle even if unnecessary
- Unchecked
  - simpler - may not handle, but avoid
  - faster
  - Error-prone - can be accidentally captured
- Rules:
  - Unchecked <= simple to avoid, local use
  - Checked <= otherwise

Are there any questions?

ROAR