

Exceptions

Ch 6

Topics

- 1) How can we handle exceptions?
- 2) How are exceptions organized?
- 3) Some convenient ways to work with exceptions.

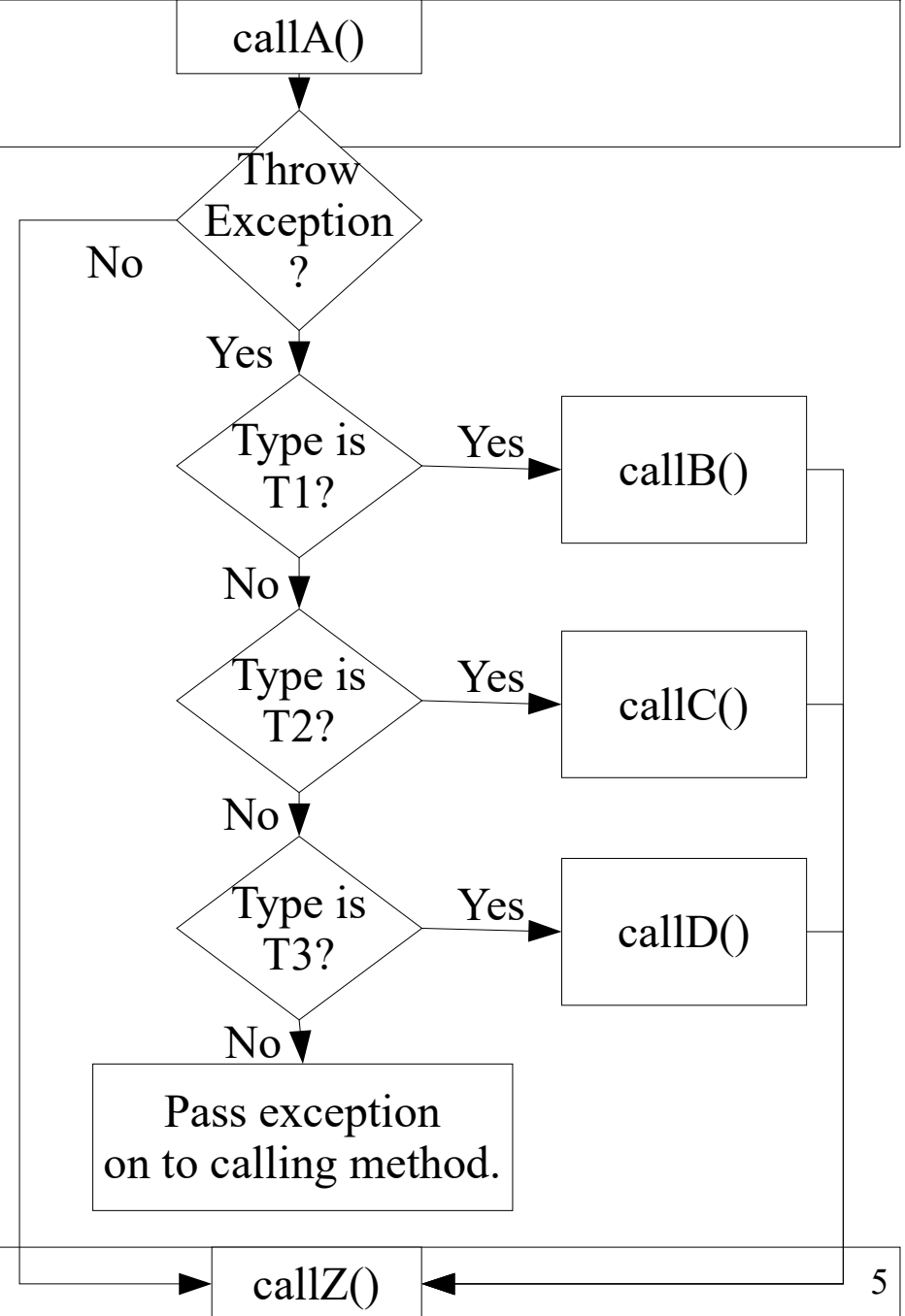
Exceptions

- Try-block: Code to check for an exception.
- Catch-block: Handle possible exceptions.
 - Check if thrown exception matches or is..

```
public static void main(String[] args) {  
    String input = "123xxx";  
    try {  
        int num = Integer.parseInt(input);  
        System.out.println("That's the number "+num);  
    } catch (NumberFormatException e) {  
        System.out.println("Bad input.");  
    }  
}
```

Try-Catch Flow

```
private void foo() {  
    try {  
        // May throw exceptions  
        callA();  
    } catch (T1 exception) {  
        callB();  
    } catch (T2 exception) {  
        callC();  
    } catch (T3 exception) {  
        callD();  
    }  
    // Some more code  
    callZ();  
}
```



Try-Catch Example

```
void tryCatch() {
    double[] data = new double[]{};

    try {
        double avg = average(data);
        System.out.println("Average value: " + avg);
    } catch (IllegalArgumentException ex) {
        System.out.println("Unable to compute: " + ex.getMessage());
    }
}

double average(double[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("Array must not be empty.");
    }

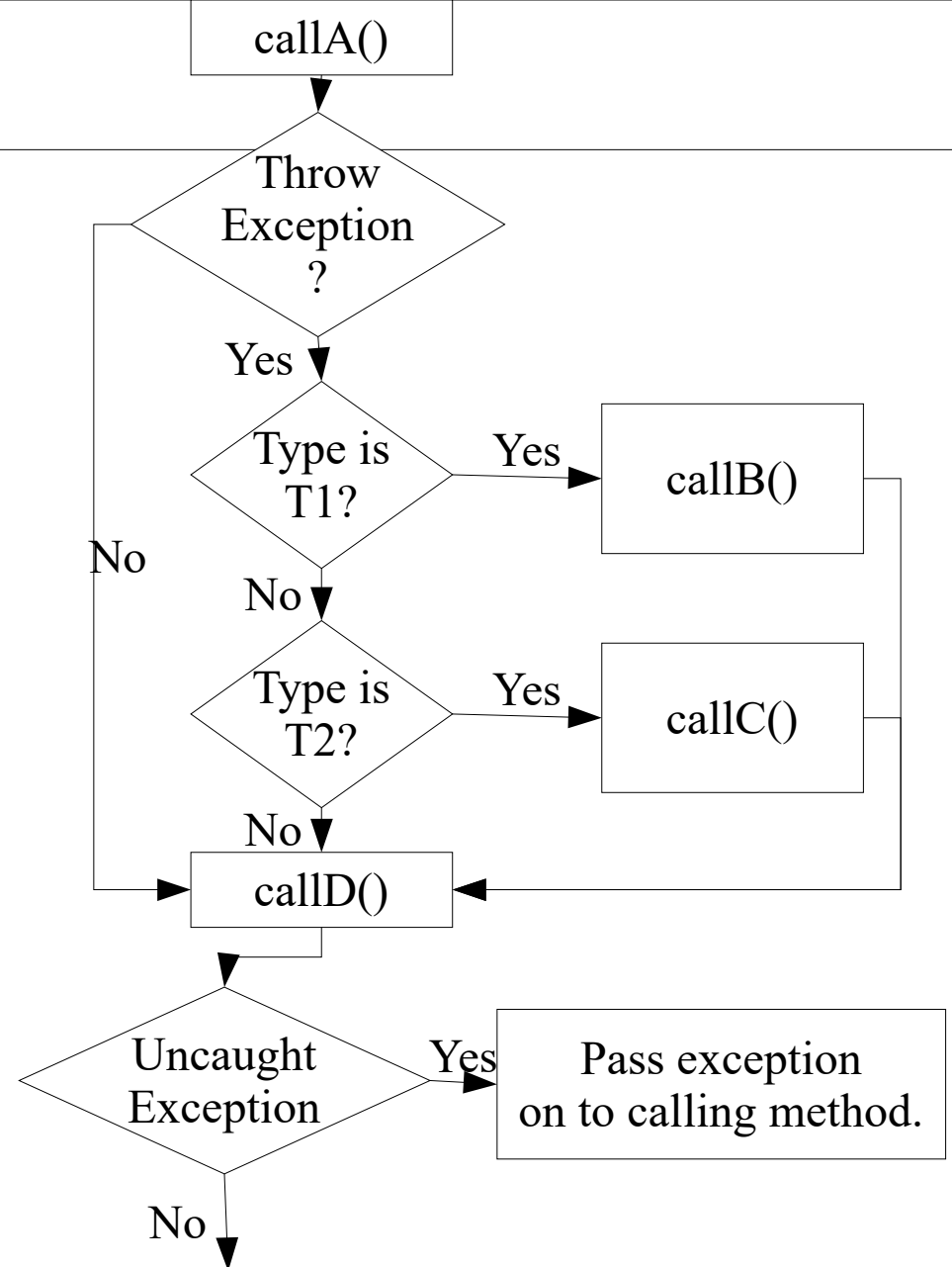
    double sum = 0;
    for (double val : data) {
        sum += val;
    }
    return sum / data.length;
}
```

The finally Clause

- Finally clause..
 - Optional clause after all the catch clauses.
- Execution Possibilities
 - No exception:
finally block executed..
 - Exception in try is un-caught:
finally block executed..
the statement which threw the exception.
 - Exception in try is caught:
finally block executed..
which catches that exception type.
- Often used for clean-up code (close file).

Try-Catch Flow

```
private void foo() {  
    try {  
        // May throw exceptions  
        callA();  
    } catch (T1 exception) {  
        callB();  
    } catch (T2 exception) {  
        callC();  
    } finally {  
        callD();  
    }  
    // Other code to do after:  
    callZ();  
}
```



Try-Finally Example

```
void tryFinally() throws IOException {
    double[] data = new double[]{};

    FileWriter fw = null;
    try {
        fw = new FileWriter("someData.txt");

        double avg = average(data);
        fw.write("Average value: " + avg);
    } finally {
        // Close the file, no matter what.
        fw.write("Encountered error... closing output file!");
        fw.close();
    }
}

double average(double[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("List must not be empty.");
    }

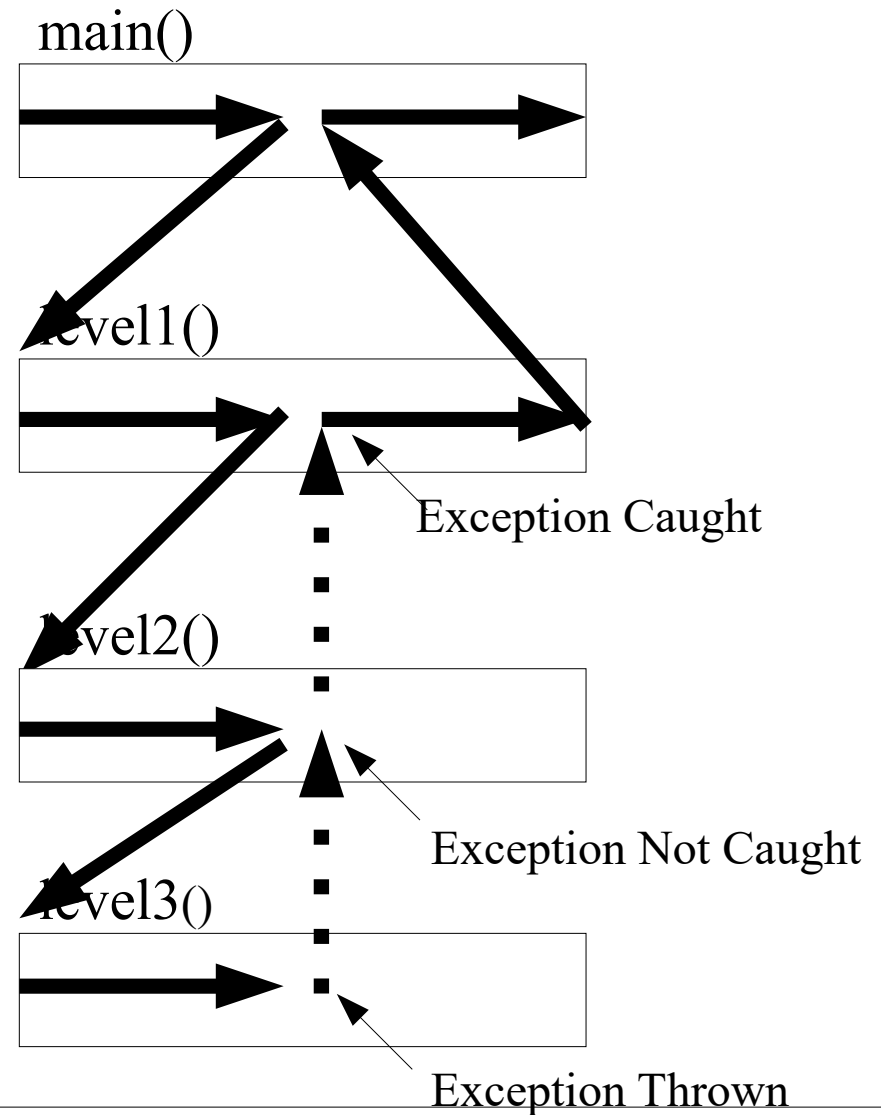
    // .... return ....
}
```


Exception Propagation

- Uncaught exception:
execution immediately..
(after finally).
 - Propagates up until a method catches exception.
 - If main() does not handle it, the program is terminated.
- Exception handling is a design decision.
 - Could..
 - Could have one of the calling methods handle it.
 - Could even let it terminate the program.
- Example:
 - Allow exception in database code..
to catch and display meaningful error.

Execution Flow

```
public class HappyCode {  
    public static void main(String[] args) {  
        level1();  
    }  
    static void level1() {  
        try {  
            level2();  
        } catch (ArithmeticException e) {  
            e.printStackTrace();  
        }  
    }  
    static void level2() {  
        level3();  
    }  
    static void level3() {  
        int a = 1 / 0;  
    }  
}
```



Re-throwing Example

```
void tryRethrow() {
    double[] grades = new double[]{};
    double avg = getAverageGrade(grades);
    System.out.println("Average grade: " + avg);
}

private double getAverageGrade(double[] grades) {
    try {
        return average(grades);
    } catch (IllegalArgumentException ex) {
        // Wrap the exception is another exception
        throw new IllegalStateException("No grades entered", ex);
    }
}

double average(double[] data) {
    if (data.length == 0) {
        throw new IllegalArgumentException("List must not be empty.");
    }

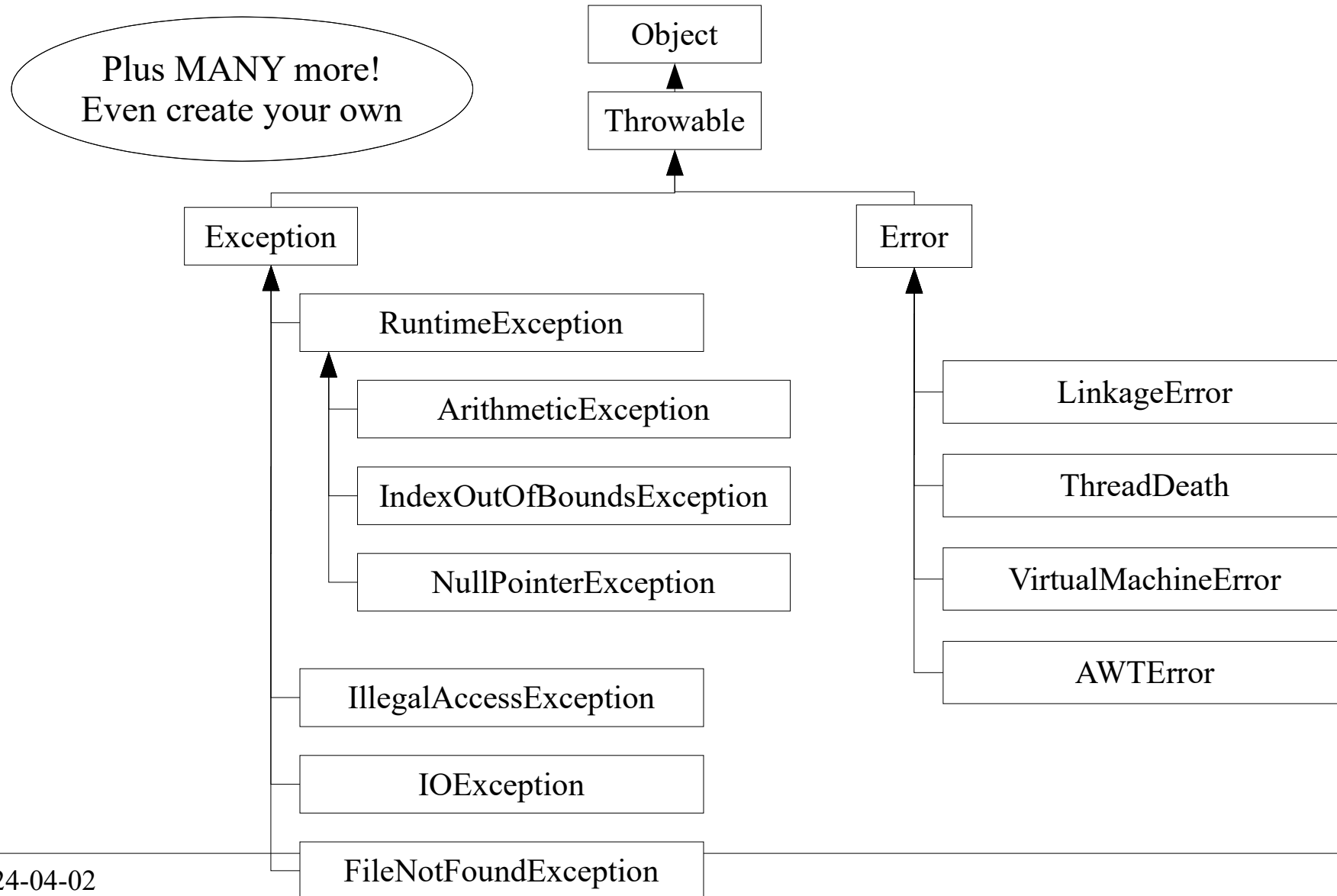
    // .... return ....
}
```

Exception Class Hierarchy

Exception Class Hierarchy

- All exceptions inherit from..
 - Many high-level exceptions in java.lang package.
 - Custom exceptions inherit from these high-level classes.
- Some methods in Exception
 - String getMessage()
Returns a string which describes the exception.
 - void printStackTrace()
Prints the stack trace to System.err (error).
 - void printStackTrace(PrintStream s)
Prints the stack trace to the given PrintStream s.

Error & Exception Hierarchy (part)



Checked vs Unchecked Exceptions

- Checked Exceptions

- Must be either..
or must be..

- This acknowledges that an exception can be thrown.

```
int foo() throws FileNotFoundException {  
    ...  
}
```

- Unchecked Exceptions

- `RuntimeException` and its subclasses are *unchecked* in throws clause.

- `RuntimeException` or its derived classes are *unchecked*
All other exceptions are *checked*.

Checked vs Unchecked

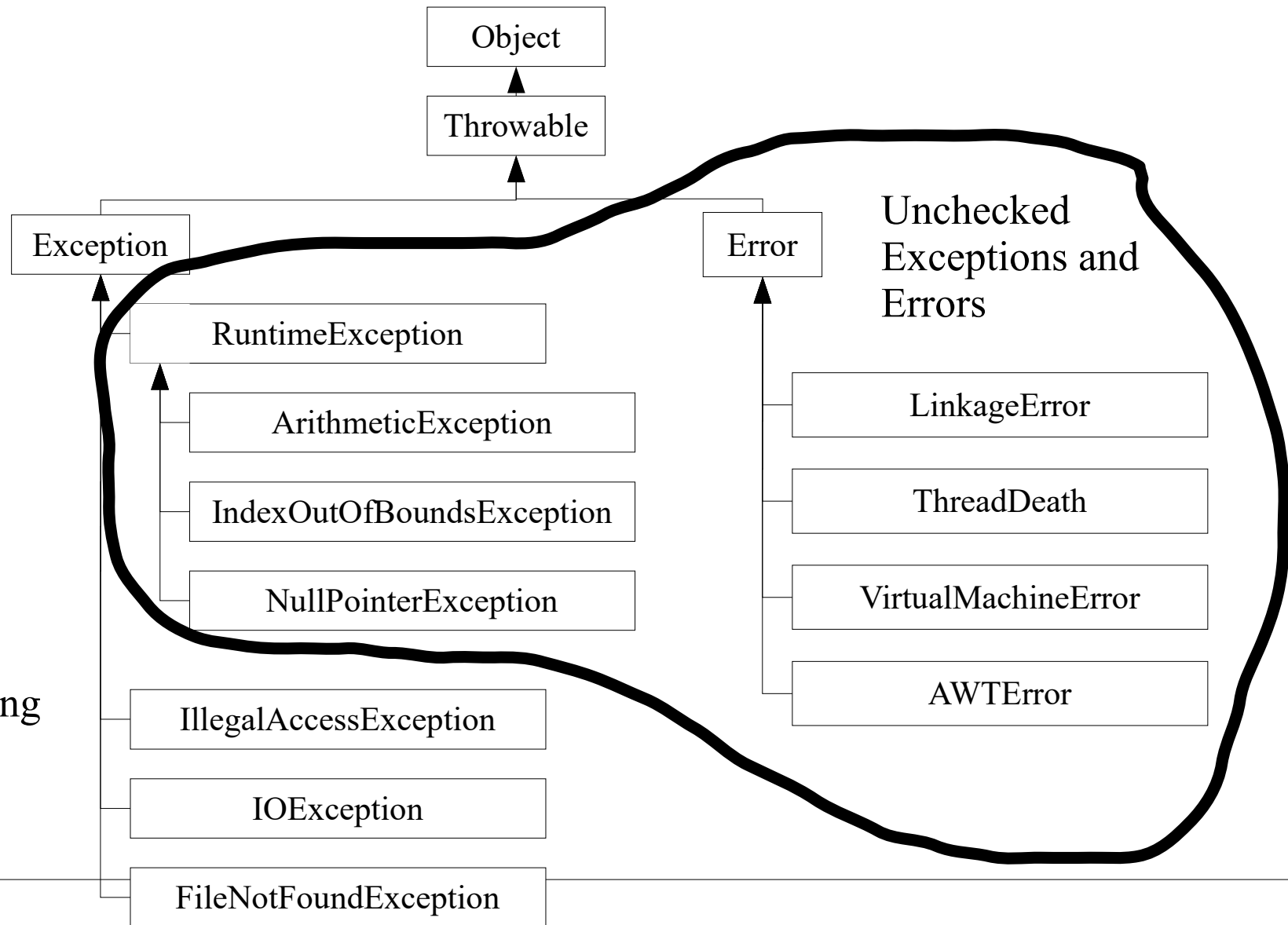
- Check vs unchecked exceptions
 - Unchecked make for cleaner..

```
public class DemoCheckedExceptions {  
    void top() throws FileNotFoundException {  
        foo1();  
    }  
    void foo1() throws FileNotFoundException {  
        foo2();  
    }  
    void foo2() throws FileNotFoundException {  
        throw new FileNotFoundException();  
    }  
}
```

```
public class DemoUncheckedException  
    void top(String[] args) {  
        foo1();  
    }  
    void foo1() {  
        foo2();  
    }  
    void foo2() {  
        throw new  
            NullPointerException();  
    }  
}
```

- Prefer unchecked exceptions
Can change which exceptions are thrown without
..

Error & Exception Hierarchy (part)



Custom Exceptions

- Create your own exceptions by..
- A new exception class allows code to specifically catch the errors your code sends.

```
/**
 * Indicates that no file was selected.
 */
public class NoFileSelected extends RuntimeException {
    public NoFileSelected() {
        super ();
    }
    public NoFileSelected(String message) {
        super (message);
    }
}
```

throw

- You can explicitly throw an exception object:

```
String getFile() {  
    if (fileName == null) {  
        throw new NoFileSelected("File not selected.");  
    }  
    return fileName;  
}
```

- As a designer, you choose..
 - throw an exception?
 - return a “failure” status (such as false or -1)?
 - try to correct the data (error recovery)

Clean Exceptions

- Exception handling can really complicate code.
- Suggestion:
 - fooThrows(): does the work, but no exception handling.
 - foo(): call's fooThrows() then does exception handling.

Original Code

```
void foo() {  
    try {  
        // do something complicated  
        // which throws exception  
        for(...) {  
            if (...)  
                throw new DaUhOh();  
        }  
    } catch (SomeException e) {  
        showUser("Oops...");  
    }  
}
```

Refactored Code

```
void foo() {  
    try {  
        fooThrows();  
    } catch (DaUhOh e) {  
        showUser("Oops...");  
    }  
}  
void fooThrows() throws DaUhOh {  
    for(...) {  
        if (...)  
            throw new DaUhOh();  
    }  
}
```

Java 7's *Exceptional* Enhancements

Resources

- Some resources must be freed:
 - Ex: Scanner's close() must be called to avoid a resource leak.
 - Can use..
- to always close resource.

```
/**
 * Read a number form a file with
 * standard try-finally
 */
public static void readNum(String fileName)
    throws FileNotFoundException
{
    Scanner scanner = null;
    File file = new File(fileName);

    try {
        scanner = new Scanner(file);
        if (scanner.hasNextInt()) {
            int num = scanner.nextInt();
            System.out.println("# " + num);
        }
    } finally {
        if (scanner != null) {
            scanner.close();
        }
    }
}
```

Try-With-Resources

- Try-with-resources

-

```
/**
 * Read a number from a file using
 * try-with-resources.
 */
public static void readNum(String fileName)
    throws FileNotFoundException
{
    File file = new File(fileName);

    try (Scanner scanner = new Scanner(file)){
        if (scanner.hasNextInt()) {
            int num = scanner.nextInt();
            System.out.println("# " + num);
        }
    }
}
```

- Can declare scanner *inside* the try ()
 - Significantly cleans up code!

- Works for objects implementing AutoClosable or Closable Interfaces.

Summary

- try-catch-finally & exception propagation.
- Checked vs unchecked exceptions
- exception inheritance hierarchy & own exceptions.
- clean exception code: unchecked & tryThrows()
- try-with-resources to close files/scanners