Exceptions

"A slipping gear could let your M203 grenade launcher fire when you least expect it. That would make you quite unpopular in what's left of your unit."

- *THE U.S. Army's PS magazine, August 1993,* quoted in *The Java Programming Language, 3rd edition*

When Good Programs Go Bad

- A variety of problems can occur when a program is running.
  - User input error: *bad url*
  - Device errors: *remote server unavailable*
  - Physical limitations: *disk full*
  - Code errors: *code that does not fulfill its contact* (i.e. pre- and post-conditions)
- When a problem occurs
  - return to safe state, save work, exit gracefully
- Code that handles a problem may be far removed from code that caused it

How to Handle Problems?

- It is possible to detect and handle problems of various types.
- Issue: this complicates the code and makes it harder to understand.
  - The problem detection and problem handling code have little or nothing to do with the *real* code is trying to do.
- A tradeoff between ensuring correct behavior under all possible circumstances and clarity of the code

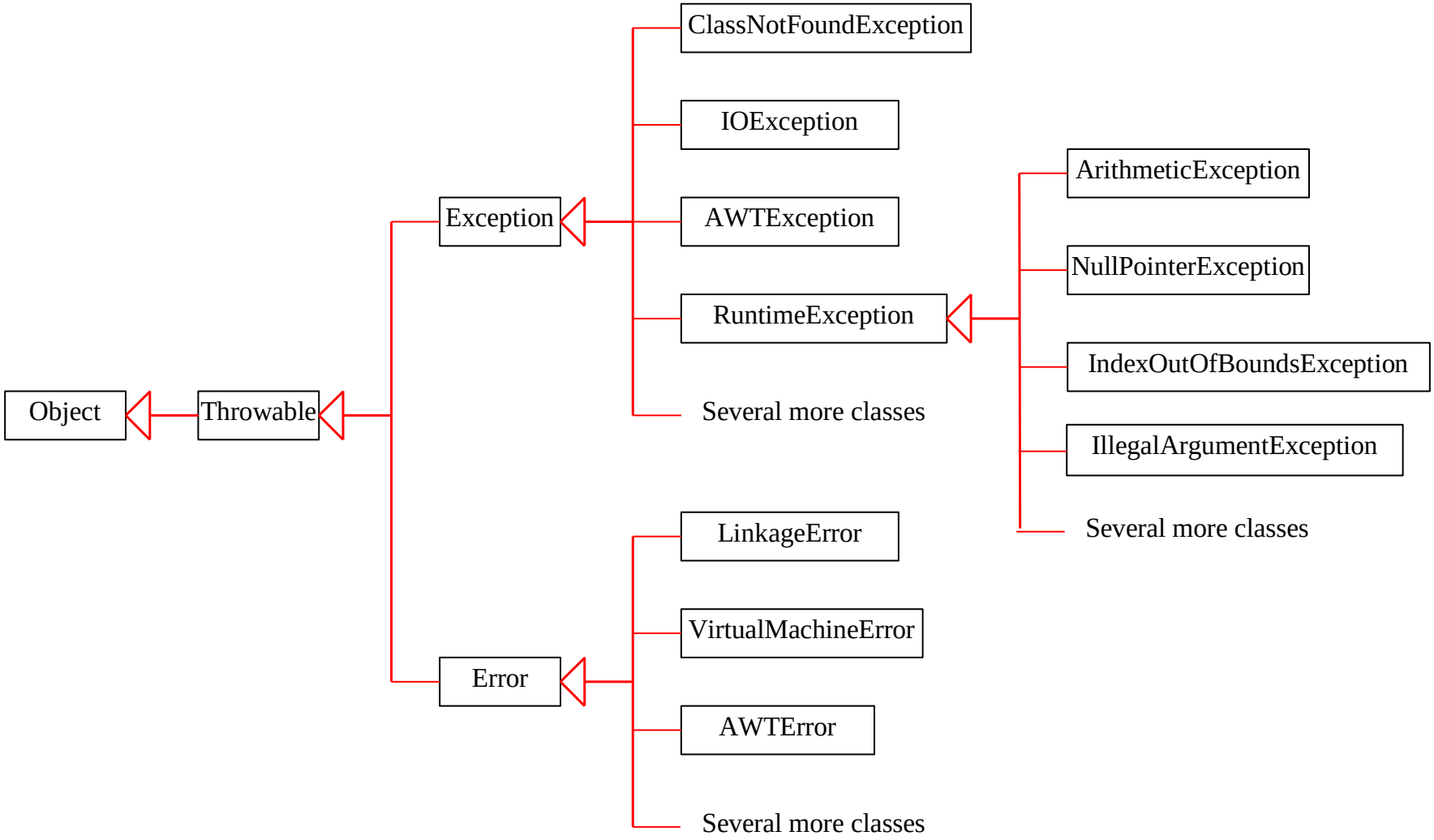
Handling Problems in Java

- Based on that of C++, but more in line with OOP philosophy
- All errors/exceptions are objects of classes that are descendants of the `Throwable` interface

Classes of Exceptions

- The Java library includes two subclasses of `Throwable` interface:
  - `Error`
    - Thrown by the Java interpreter for events such as heap overflow
    - Never handled by user programs
  - `Exception`
    - User-defined exceptions are usually subclasses of this
    - Has two predefined subclasses:
      - `IOException`
      - `RuntimeException`

Error/Exception Classes



Throwable Interface Methods

Methods with Description
public String getMessage() Returns a detailed message about the exception that has occurred. This message is initialized in the `Throwable` constructor.
public Throwable getCause() Returns the cause of the exception as represented by a `Throwable` object.
public String toString() Returns the name of the class concatenated with the result of getMessage()
public void printStackTrace() Prints the result of `toString()` along with the stack trace to `System.err`, the error output stream.
public StackTraceElement [] getStackTrace() Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
public Throwable fillInStackTrace() Fills the stack trace of this `Throwable` object with the current stack trace, adding to any previous information in the stack trace.

Exceptions

- Many languages, including Java, use a mechanism know as exceptions to handle problems at runtime
  - In Java, `Exception` is a class with many descendants.
  - For example:
    - `ArrayIndexOutOfBoundsException`
    - `NullPointerException`
    - `FileNotFoundException`
    - `ArithmeticException`
    - `IllegalArgumentException`

Handling Exceptions

- Exceptions in Java fall into two different categories:
  - *Checked (not Runtime) and Unchecked (Runtime)*
  - *Checked* exception
    - Must be caught in `catch` block
    - Or declared in `throws` clause
  - *Unchecked* exception
    - Need not be caught in `catch` block or declared in `throws`
    - Exceptions that exist in code should be fixed

Unchecked Exceptions

- Unchecked exceptions are ***completely preventable*** and should never occur.
  - Caused by logic errors, created by us, the programmers.
  - Descendents of the RuntimeException class
- There does not *need* to be special error handling code
  - Just regular error prevention code
- If error handling code was required, programs would be unwieldy

Unchecked Runtime Exceptions

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Checked Exceptions

- "Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur, and if they do occur must be dealt with in some way.[other than the program terminating.]"
  - *Java Programming Language, Third Edition*
- Checked exceptions represent errors that are unpreventable by us

Checked Exceptions

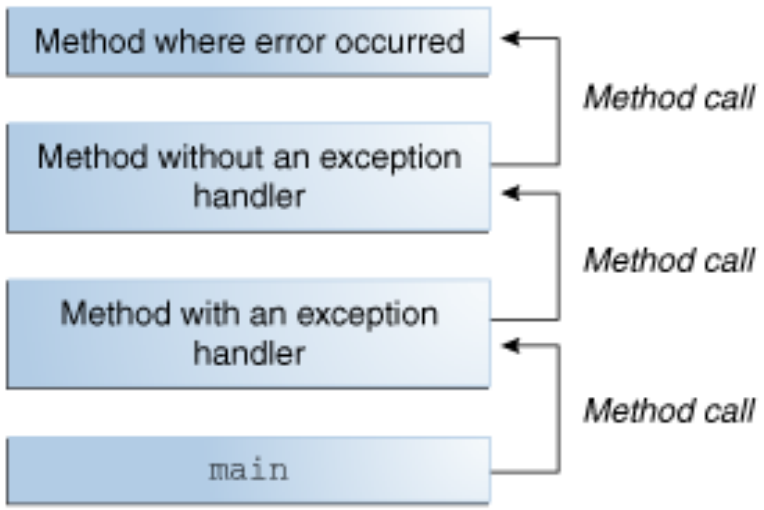
Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

When Exceptions Occur

- When an exception occurs, the normal flow of control of a program halts
- The method where the exception occurred creates an exception object of the appropriate type
  - The object contains information about the error
  - Then hands the exception to the Java Runtime System
- The Java Runtime System (JRS) searches for a matching `catch` block for the exception
- The first matching `catch` block is executed
- When the `catch` block code is completed, the program goes the next regular statement after the `catch` block

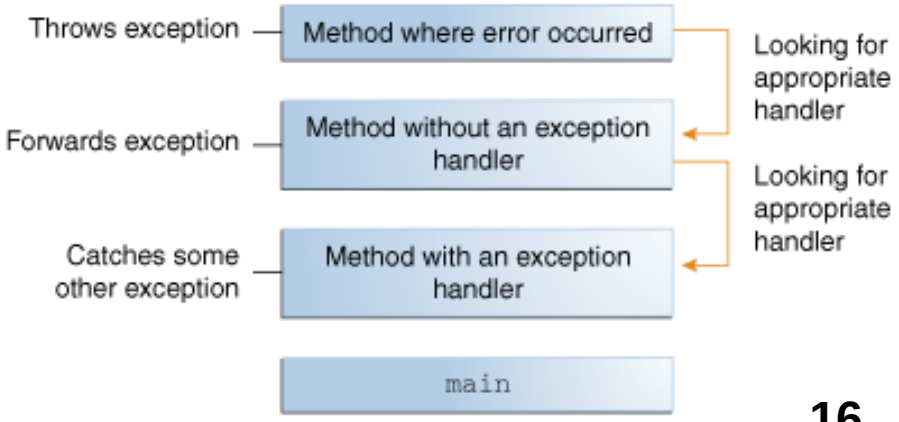
Call Stack

- After a method throws an exception, the JRS looks for a method with a `catch` block to handle it. This block of code is called the *exception handler*. If the method that threw the exception can't handle it, the JRS looks in the ordered list of methods that have been called to get to this method. This list of methods is known as the *call stack*.



Exception Handler

- The search for an exception handler starts with the method where the error occurred. The search then proceeds down the call stack. When an appropriate handler is found, the JRS passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.



Catching Exceptions

- The exception handler chosen is said to *catch the exception*. If the JRS the whole call stack without finding an appropriate exception handler, the JRS and the program terminate.

Catching Exceptions

- Method uses a `try` block around code that may cause an exception.

```
try
{
      //Protected code
}
catch(ExceptionName e1)
{
      //Catch block
}
```

- Catch statement declares type of exception trying to catch. If an exception occurs of that type, the exception is passed on to that block for processing.

An Example

```
// File Name : ExcepTest.java

public class ExcepTest
{
   public static void main(String args[])
   {

      try
      {
            int a[] = new int[2];
            System.out.println("Access element three :" +
a[3]);
      }
      catch(ArrayIndexOutOfBoundsException e)
      {
            System.out.println("Exception thrown :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

Example – cont'd

- The above code would produce the following result:

```
Exception thrown
:java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple `Catch` Blocks

- A `try` block can be followed by multiple `catch` blocks.

```
try
{
        //Protected code
}
catch(ExceptionType1 e1)
{
        //Catch block
}
catch(ExceptionType2 e2)
{
        //Catch block
}
catch(ExceptionType3 e3)
{
        //Catch block
}
```

Example-Multiple Catch Blocks

```
try
{
      file = new
          FileInputStream(fileName);
      x = file.read();
}
catch(FileNotFoundException f)
{
      f.printStackTrace();
}
catch(IOException i)
{
      i.printStackTrace();
}
```

`throws` / `throw` Keywords

- If a method does not handle a *checked* exception, the method must declare it using the `throws` keyword.

```
import java.io.*;
public class ClassName
{
      public int read(FileStream file) throws
IOException
      {
          int x = file.read();
      }
}
```

Multiple Exceptions Thrown

- A method can throw more than one exception
  - Exceptions are declared in a list separated by commas.

```
import java.io.*;
public class ClassName
{
  public int read(String fileName) throws
FileNotFoundException, IOException
  {
      FileStream file = new FileInputStream(filename);
      int x file.read();
  }
}
```

`finally` Block

- The `finally` keyword used to create a block of code that follows a try block.
  - always executes, whether or not an exception has occurred.
  - run any cleanup-type statements that you want to execute
  - appears after the `catch` blocks

```
try
{ //Protected code
}
catch(ExceptionType e)
{ //Catch block
}
finally
{ //The finally block always executes.
}
```

Example – `finally` Block

```
public class ExcepTest
{
  public static void main(String args[])
  {
        int a[] = new int[2];
        try
        {
          System.out.println("Access element three :" + a[3]);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
          System.out.println("Exception thrown :" + e);
        }
        finally
        {
          a[0] = 6; System.out.println("First element value: " +a[0]);
          System.out.println("The finally statement is executed");
        }
  }
}
```

`finally` Block Example – cont'd

- Would produce this result:

```
Exception thrown
:java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Things to Note

- Must follow a `try` block with either `catch` and/or `finally` block.
- `Catch` blocks must follow a `try` statement.
- A `finally` block is not required if there are `try-catch` blocks.
- `try`, `catch`, `finally` blocks must follow each other in that order
  - There cannot be any code between them.

Throwing Exceptions Yourself

- If you wish to throw an exception in your code, you use the `throw` keyword
- Most common would be for an unmet precondition

```
public class Circle
{     private double radius;

      // other variables and methods

      public setRadius(int radius){
            if (radius <= 0)
                  throw new IllegalArgumentException
                        ("radius must be > 0. "
                        + "Value of radius: " + radius);
            this.radius = radius;
            // other unrelated code
      }
}
```

Creating Exceptions

- All exceptions must inherit from a class that implements the `Throwable` interface.
- If you want to write a checked exception, extend the `Exception` class.
- If you want to write a runtime exception, extend the `RuntimeException` class.
- Define at least two constructors
  1. Default, no parameters
  2. Other with `String` parameter
  - For both, call constructor of base class using `super`
- Do not override inherited `getMessage`

Example–Creating Exceptions

```
// File Name MyException.java
import java.io.*;
public class MyException extends Exception
{
     public MyException(String msg)
     {
          super(msg);
     }
}
```

Errors

- An *error* is an object of class `Error`
  - Similar to an unchecked exception
  - Need not catch or declare in throws clause
  - Object of class `Error` generated when abnormal conditions occur
- `Error`s are more or less beyond your control
  - Require change of program to resolve

Question 1

What is output by the method `badUse` if it is called with the following code?

```
     …
int[] nums = {3, 2, 6, 1};
badUse( nums );
     …

public static void badUse(int[] vals){
    int total = 0;
    try{
        for(int i = 0; i < vals.length; i++){
            int index = vals[i];
            total += vals[index];
        }
   }
    catch(Exception e){
        total = -1;
    }
    System.out.println(total);
}
```

A. 12 B. 0 C. 3 D. −1 E. 5

Question 1

What is output by the method `badUse` if it is called with the following code?

```
     …
int[] nums = {3, 2, 6, 1};
badUse( nums );
     …

public static void badUse(int[] vals){
    int total = 0;
    try{
        for(int i = 0; i < vals.length; i++){
            int index = vals[i];
            total += vals[index];
        }
   }
    catch(Exception e){
        total = -1;
    }
    System.out.println(total);
}
```

A. 12 B. 0 C. 3 D. −1 E. 5

Question 2

Is the use of a `try-catch` block on the previous question a proper use of `try-catch` blocks?

A. Yes

B. No

Question 2

Is the use of a `try-catch` block on the previous question a proper use of `try-catch` blocks?

A. Yes

B. No

What is a better way to handle this type of error?

Advantages of Exceptions

1. Separating Exception-Handling Code from "Regular" Code
   - Means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
2. Propagating Exceptions Down the Call Stack
   - Ability to propagate error reporting down the call stack of methods.
3. Grouping and Differentiating Error Types
   - Since all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.

Assertions

- Statements in the program declaring a Boolean expression about the current state of variables
- If evaluate to `true`, nothing happens
- If evaluate to `false`, an `AssertionError` exception is thrown
- Can be disabled during runtime without program modification or recompilation
- Two forms
  - `assert` *condition*`;`
  - `assert` *condition*`:` *expression*`;`

Error Handling, Error Handling Everywhere!

- Seems like a lot of choices for error prevention and error handling
  - normal program logic (e.g. `if`s, `for`-loop counters)
  - assertions
  - `try-catch` block
- When is it appropriate to use each kind?

Error Prevention

- Use program logic (`if`s , `for`s) to prevent logic errors and unchecked exceptions
  - dereferencing a null pointer
  - going outside the bounds of an array
  - violating the preconditions of a method you are calling

Error Prevention – cont'd

- In general, don't use asserts to check preconditions
  - Standard Java style is to use exceptions
- Use `try-catch` blocks on checked exceptions
  - Don't use them to handle unchecked exceptions, like null pointer or array index out of bounds
- One place it is reasonable to use `try-catch` is in testing suites.
  - put each test in a `try-catch`. If an exception occurs that test fails, but other tests can still be run