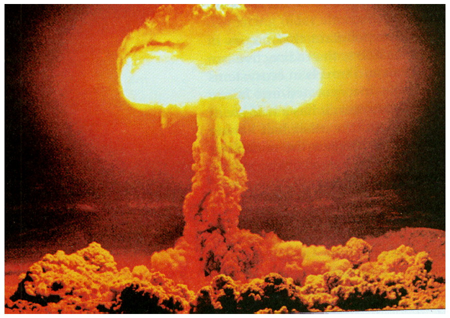
CSE 331

Exceptions and Error-Handling

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Exceptions



- **exception**: An object representing an error.
  - Other languages don't have this concept; they represent errors by returning error codes (null, -1, false, etc.).
    - Are exceptions better? What are their benefits?

- **throw**: To cause an exception to occur.
  - What are some actions that commonly throw exceptions?

- **catch**: To handle an exception.
  - If an exception is thrown and no code catches it, the program's execution will stop and an error trace will be printed.
  - If the exception is caught, the program can continue running.

Code that throws exceptions

- dividing by zero:

```
int x = 0;
System.out.println(1 / x);  // ArithmeticException
```

- trying to dereference a null variable:

```
Point p = null;
p.translate(2, -3);         // NullPointerException
```

- trying to interpret input in the wrong way:

```
// NumberFormatException
int err = Integer.parseInt("hi");
```

- reading a non-existent file:

```
// FileNotFoundException
Scanner in = new Scanner(new File("notHere.txt"));
```

Exception avoidance

- In many cases, the best plan is to try to avoid exceptions.

```
// better to check first than try/catch without check
int x;
...
if (x != 0) {
    System.out.println(1 / x);
}


File file = new File("notHere.txt");
if (file.exists()) {
    Scanner in = new Scanner(file);
}


// can we avoid this one?
int err = Integer.parseInt(str);
```

Catching an exception

```
try {
    statement(s);
} catch (type name) {
    code to handle the exception
}
```

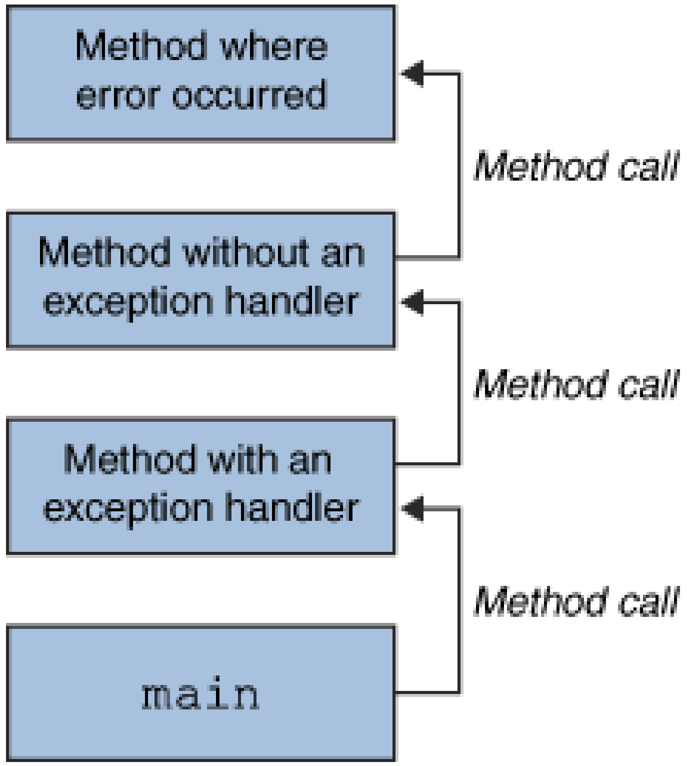
- The `try` code executes. If the given exception occurs, the `try` block stops running; it jumps to the `catch` block and runs that.

```
try {
    Scanner in = new Scanner(new File(filename));
    System.out.println(input.nextLine());
} catch (FileNotFoundException e) {
    System.out.println("File was not found.");
}
```

Throwing and catching

- At any time, your program has an active **call stack** of methods.

- When an exception is thrown, the JVM looks up the call stack until it finds a method with a matching `catch` block for it.
  - If one is found, control jumps back to that method.
  - If none is found, the program crashes.


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

- Exceptions allow **non-local error handling**.
  - A method many levels up the stack can handle a deep error.

Catch, and then what?

```
public void process(String str) {
    int n;
    try {
        n = Integer.parseInt(str);
    } catch (NumberFormatException nfe) {
        System.out.println("Invalid number: " + str);
    }
    ...
```

- Possible ways to handle an exception:
  - retry the operation that failed
  - re-prompt the user for new input
  - print a nice error message
  - quit the program
  - do nothing (!)   (why? when?)

Exception methods

- All exception objects have these methods:

Method	Description
`public String` **`getMessage`**`()`	text describing the error
`public String` **`toString`**`()`	exception's type and description
`getCause``(),` **`getStackTrace`**`(),` **`printStackTrace`**`()`	other methods

```
try {
    readFile();
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
}
```

Design and exceptions

- **Effective Java Tip #57:**
  Use exceptions only for exceptional conditions.
  - The author of the `Integer.parseInt` method got this wrong.
  - Strings that are not legal as `int`s are common (not "exceptional").
    - (What should they have done instead?)

```
// Can we avoid this one?  Not really.  :-(
int n;
try {
    n = Integer.parseInt(str);
} catch (NumberFormatException nfe) {
    n = -1;
}
```

Ignoring exceptions

- **Effective Java Tip #65:** Don't ignore exceptions.
  - An empty `catch` block is (a common) poor style.
    - often done to get code to compile or hide an error

```
try {
    readFile(filename);
} catch (IOException e) {}  // do nothing on error
```

  - At a *minimum*, print out the exception so you know it happened.

```
} catch (IOException e) {
    e.printStackTrace();    // just in case
}
```

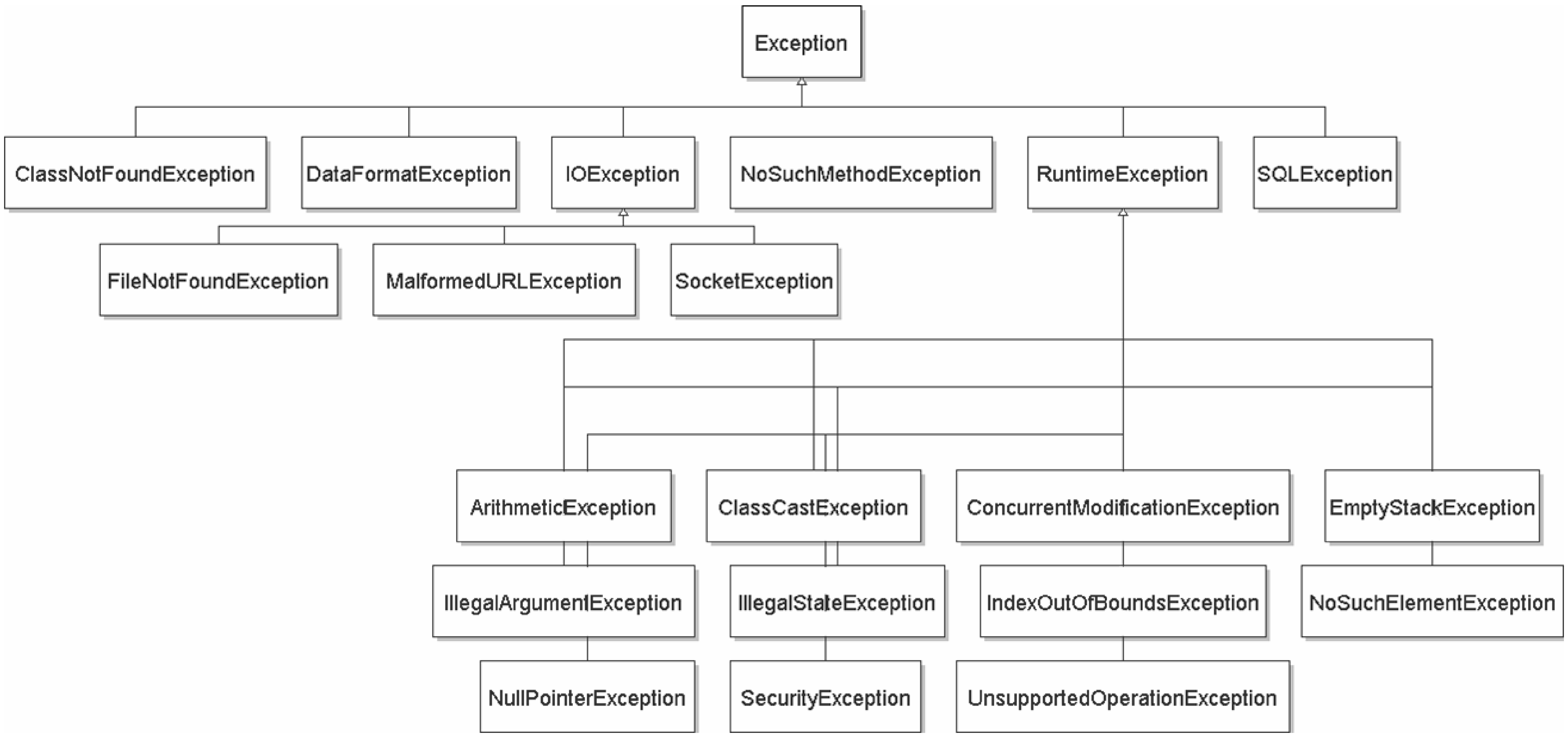
Catch multiple exceptions

```
try {
      statement(s);
} catch (type1 name) {
      code to handle the exception
} catch (type2 name) {
      code to handle the exception
 ...
} catch (typeN name) {
      code to handle the exception
}
```

- You can catch more than one kind of exception in the same code.
- When an exception is thrown, the matching catch block (if any) is used.
- If multiple `catch` blocks match, the most specific match is chosen.

Exception inheritance

- All exceptions extend from a common superclass `Exception`



Some common exceptions

- ArithmeticException
- BufferOverflowException
- ClassCastException
- ClassNotFoundException
- CloneNotSupportedException
- ConcurrentModificationException
- EmptyStackException
- IllegalArgumentException
- IllegalStateException
- IndexOutOfBoundsException
- InterruptedException
- IOException
  - EOFException, FileNotFoundException, InterruptedIOException, MalformedURLException, ... NotSerializableException, SocketException, SSLException, UnknownHostException, ZipException
- JarException
- MalformedURLException
- NegativeArraySizeException
- NoSuchElementException
- NullPointerException
- ProtocolException
- RuntimeException
- SecurityException
- UnknownElementException
- UnsupportedOperationException

- see also: http://mindprod.com/jgloss/exception.html

Inheritance and exceptions

- You can catch a general exception to handle any subclass:

```
try {
    Scanner input = new Scanner(new File("foo"));
    System.out.println(input.nextLine());
} catch (Exception e) {
    System.out.println("File was not found.");
}
```

- Similarly, you can state that a method throws any exception:

```
public void foo() throws Exception { ...
```

  - Are there any disadvantages of doing so?

Catching with inheritance

```
try {
    statement(s);
} catch (FileNotFoundException fnfe) {
    code to handle the file not found exception
} catch (IOException ioe) {
    code to handle any other I/O exception
} catch (Exception e) {
    code to handle any other exception
}
```

- a `SocketException` would match the second block
- an `ArithmeticException` would match the third block

Who should catch it?

- The code that is able to handle the error properly should be the code that catches the exception.
  - Sometimes this is not the top method on the stack.

- Example:
  - main → showGUI() → click() → readFile() → FileNotFoundException!
    - Which method should handle the exception, and why?
  - main → new PokerGame() → new Player() → loadHistory() → Integer.parseInt() -> NumberFormatException
    - Which method should handle the exception, and why?

Throwing an exception

`throw new` **ExceptionType**`("`**message**`");`

- It is common practice to throw exceptions on unexpected errors.

```
public void deposit(double amount) {
    if (amount < 0.0) {
        throw new IllegalArgumentException();
    }
    balance += amount;
}
```

  - Why throw rather than just ignoring the negative value?
    - Why not return a special error code, such as `-1` or `false`?

Good throwing style

- An exception can accept a `String` parameter for a message describing what went wrong.
  - This is the string returned by `getMessage` in a `catch` block.

```
public void deposit(double amount) {
    if (amount < 0.0) {
        throw new IllegalArgumentException(
                         "negative deposit: " + amount);
    }
    balance += amount;
}
```

- **EJ Tip #63:** Include failure-capture information in detail messages.
  - Tell the caller what went wrong, to help them fix the problem.

Commenting exceptions

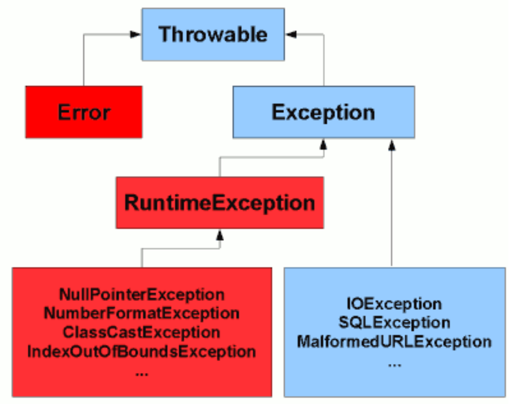
- If your method throws, *always* explain this in the comments.
  - State the types of exceptions thrown and under what conditions.

```
// Places the given amount of money into this account.
// Throws an IllegalArgumentException on negative deposits.
public void deposit(double amount) {
    if (amount < 0.0) {
        throw new IllegalArgumentException(
                         "negative deposit: " + amount);
    }
    balance += amount;
}
```

- **EJ Tip #62:** Document all exceptions thrown by each method.
  - The client must know this in order to avoid or catch the exceptions.

Checked exceptions

- Java has two major kinds of exceptions:
  - **checked exceptions:** Ones that MUST be handled by a `try/catch` block (or `throws` clause) or else the program will not compile.
    - Meant for serious problems that the caller ought to deal with.
    - Subclasses of `Exception` in the inheritance tree.
  - **runtime exceptions:** Ones that don't have to be handled; if not handled, the program halts.
    - Meant for smaller errors or programmer errors.
    - Subclasses of `RuntimeException` in the tree.
    - Mistakes that could have been avoided by a test.
      - check for null or 0, check if a file exists, check array's bounds, ...



The throws clause

```
public type name(parameters) throws type {
```

- A clause in a method header claiming it may cause an exception.
  - Needed when a method may throw an uncaught checked exception.

```
public void processFile(String filename)
                  throws FileNotFoundException {
```

  - The above means one of two possibilities:
    - `processFile` itself might throw an exception.
    - `processFile` might call some sub-method that throws an exception, and it is choosing not to catch it (rather, to re-throw it out to the caller).

Writing an exception class

- **EJ Tip #61:** Throw exceptions appropriate to the abstraction.
  - When no provided exception class is quite right for your app's kind of error, you should write your own `Exception` subclass.

```
// Thrown when the user tries to play after the game is over.
public class GameOverException extends RuntimeException {
    private String winner;

    public GameOverException(String message, String winner) {
        super(message);
        this.winner = winner;
    }

    public String getWinner() {
        return winner;
    }
}

// in Game class...
if (!inProgress()) {
    throw new GameOverException("Game already ended", winner);
```

Checked exceptions suck!

- **EJ Tip #59:** Avoid unnecessary use of checked exceptions.
  - Checked exceptions are (arguably) a wart in the Java language.
  - It should be the client's decision whether or not to catch exceptions.
  - When writing your own exception classes, extend `RuntimeException` so that it doesn't need to be caught unless the client wants to do so.
    - Some cases still require throwing checked exceptions (e.g. file I/O)

```
public void play() throws Exception {             // no
public void play() throws RuntimeException {    // better
public void play() throws MP3Exception {         // best

public class MP3Exception extends RuntimeException { ... }
```

Problem: redundant code

```
public void process(OutputStream out) {
    try {
        // read from out;  might throw
        ...
        out.close();
    } catch (IOException e) {
        out.close();
        System.out.println("Caught IOException: "
                           + e.getMessage());
    }
}
```

- The close code appears redundantly in both places.
- Can't move it out below the `try/catch` block because `close` itself could throw an `IOException`.

The finally block

```
try {
    statement(s);
} catch (type name) {
    code to handle the exception
} finally {
    code to run after the try or catch finishes
}
```

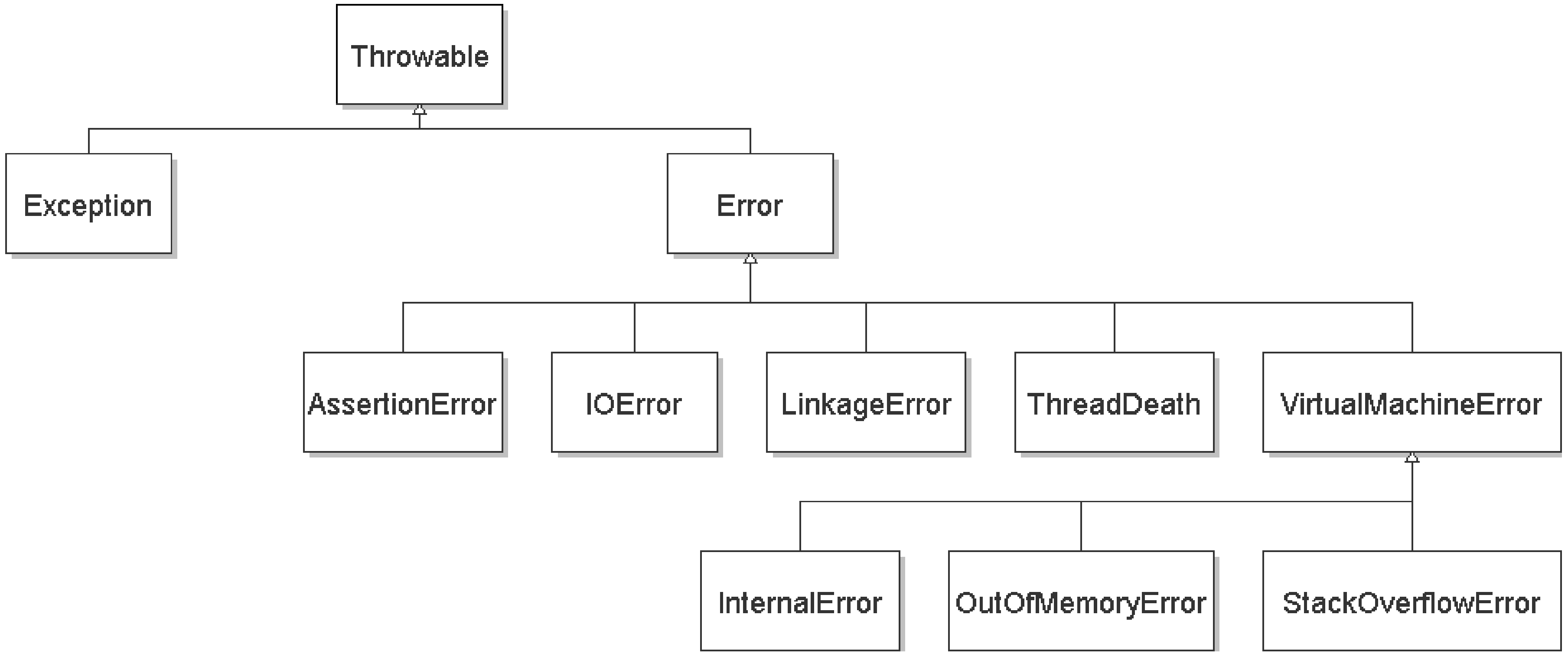
- `finally` is often used for common "clean-up" code.

```
try {
    // ... read from out;  might throw
} catch (IOException e) {
    System.out.println("Caught IOException: "
                       + e.getMessage());
} finally {
    out.close();
}
```

  - The `catch` block is optional; `try/finally` is also legal.

Exceptions and errors

- There are also `Errors`, which represent serious Java problems.
  - `Error` and `Exception` have common superclass `Throwable`.
  - You can catch an `Error` (but you probably shouldn't)



Common errors

- AbstractMethodError
- AWTError
- ClassFormatError
- ExceptionInInitializerError
- IllegalAccessError
- InstantiationError
- InternalError
- LinkageError
- NoClassDefFoundError
- NoSuchFieldError
- NoSuchMethodError
- OutOfMemoryError
- ServerError
- StackOverflowError
- UnknownError
- UnsatisfiedLinkError
- UnsupportedClassVersionError
- VerifyError
- VirtualMachineError