Exception Handling

Introduction

- An *exception* is an abnormal condition that arises in a code sequence.
- An exception can occur for many different reasons:
  - A user has entered invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
- An Error "indicates serious problems that a reasonable application should not try to catch."

while

- An Exception "indicates conditions that a reasonable application might want to catch."

Error Handling

- Instead of programming for success

```
x.doSomething()
```

- You would always be programming for failure:

```
if (!x.doSomething()) return false;
```

- Normally programs cannot recover from errors.
  - E.g. JVM is out of Memory.

- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes.

Throwing Exceptions

- Exceptions:
  - Can't be overlooked
  - Sent directly to an exception handler–not just caller of failed method
- Throw an exception object to signal an exceptional condition
- Example: `IllegalArgumentException:`

```
illegal parameter value
IllegalArgumentException exception
   = new IllegalArgumentException("Amount exceeds balance");
throw exception;
```

Continued…

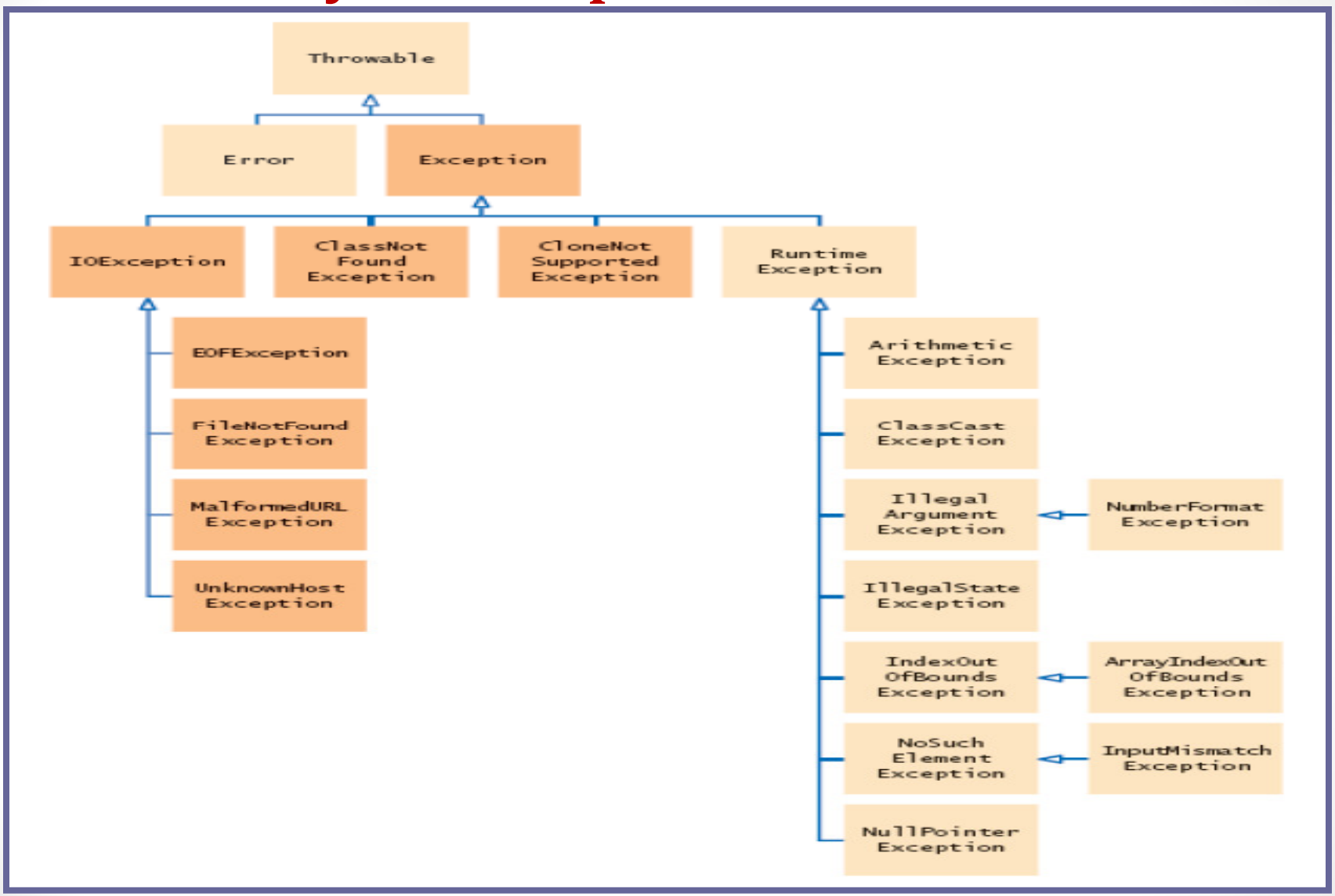
Throwing Exceptions

- No need to store exception object in a variable:

```
throw new IllegalArgumentException("Amount exceeds balance");
```

- When an exception is thrown, method terminates immediately
  - Execution continues with an exception handler

Hierarchy of Exception Classes



The Hierarchy of Exception Classes

Checked and Unchecked Exceptions

- Two types of exceptions:
  - Checked
    - The compiler checks that you don't ignore them.
    - A user error or a problem that cannot be foreseen by the programmer.
    - Cannot be ignored at the time of compilation.
    - Majority occur when dealing with input and output.
    - For example, `IOException`

Checked and Unchecked Exceptions

- Unchecked:
  - Extend the class `RuntimeException or Error.`
  - Ignored at the time of compilation.

  RuntimeException:
  - They are the programmer's fault.
  - Examples of runtime exceptions:
  - E.g. division by zero and invalid array indexing
  - Error:
  - Problems that arise beyond the control of the user or the programmer.
  - Example of error: `OutOfMemoryError`

Uncaught Exceptions

- Any exception that is not caught by your program will ultimately be processed by the default handler provided by Java Runtime System.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

```
class Exc0 {
public static void main(String args[]) {
        int d = 0;
        int a = 42 / d;
        }
}
```

Output:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Keywords

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed before a method returns is put in a **finally** block.

Syntax

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
finally {
// block of code to be executed before try block ends
}
```

Using try and catch

- Y to handle exception when default handler does that?
- Two benefits:
  - First, it allows you to fix the error.
  - Second, it prevents the program from automatically terminating.

```
class Exc2 {
public static void main(String args[]) {
        int d, a;
        try { // monitor a block of code.
        d = 0;
        a = 42 / d;
        System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
        System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
        }
}
```

Output:
Division by zero.
After catch statement.

- Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.

Multiple catch Clauses

- MultiCatch.java
- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses.
  - because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
  - Thus, a subclass would never be reached if it came after its superclass.
  - Generates compile time unreachable code error.
  - SuperSubCatch.java

Catching Exceptions

- Statements in try block are executed.
- If no exceptions occur, `catch` clauses are skipped
- If exception of matching type occurs, execution jumps to `catch` clause.
- If exception of another type occurs, it is thrown until it is caught by another `try` block

Nested try Statements

- MethNestTry.java

throw

- Till now you have been catching exceptions that are thrown by the Java run-time system.
- Program can throw an exception explicitly, using the **throw** statement.
  - throw *e*;
- *e* must be an object of type **Throwable** or a subclass of **Throwable**.
- Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator. ThrowDemo.java

throws

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

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

- ThrowsDemo.java

finally

- The finally keyword is used to create a block of code that follows a try block.
- A finally block of code always executes, whether or not try block throw an exception.
- Useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. FinallyDemo.java

Con't

Note:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
  - i.e. The **finally** clause is optional.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Throwable class 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.

Unchecked Runtime Exceptions Subclasses

Exception	Meaning
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.

Con't

Exception	Meaning
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 defined in java.lang

Exception	Meaning
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.

Example of Checked Exception

- Example.java
- Example1.java

Creating user defined exception class

- All exceptions must be a child of Throwable.
- If you want to write a checked exception need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.
- BankDemo.java

Chained Exceptions

- To allow chained exceptions, Java 2, version 1.4 added two constructors and two methods to **Throwable**.
- Throwable(Throwable *causeExc*)
  - *causeExc* is the exception that causes the current exception.
- Throwable(String *msg*, Throwable *causeExc*)
- Throwable getCause( )
- Throwable initCause(Throwable *causeExc*)
  - ChainExcDemo.java