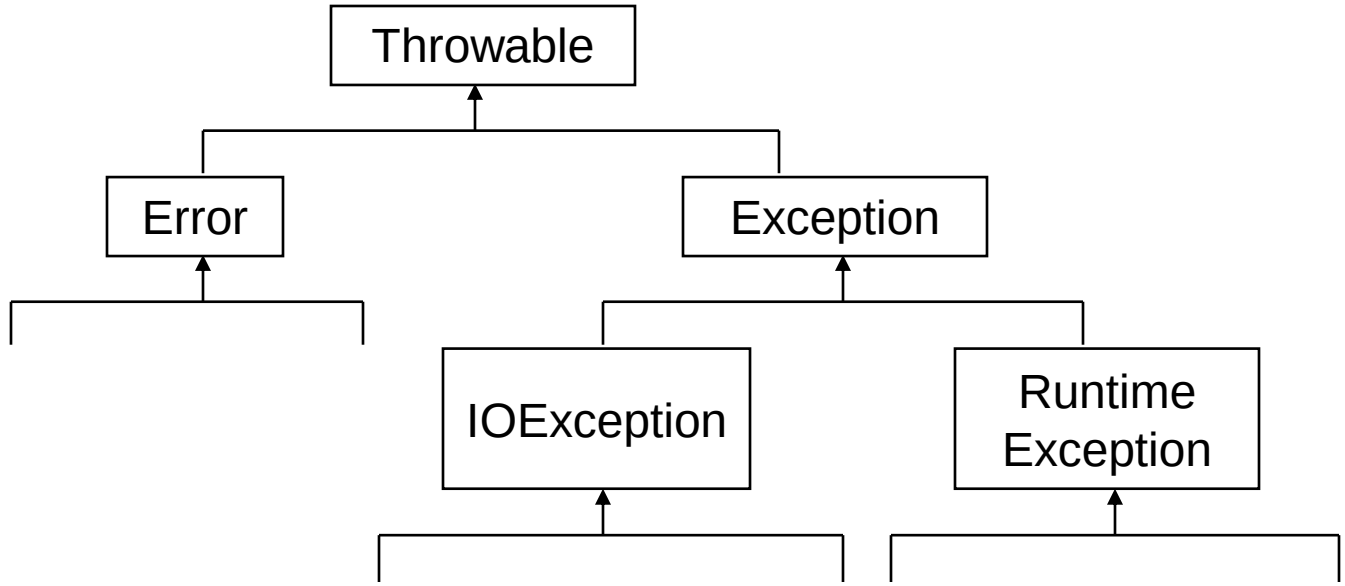
Exceptions

By Pakita Shamoi, Fall 2012

Overview of the exception hierarchy

A simplified diagram of the exception hierarchy in Java



- All exceptions extend the class `Throwable`, which immediately splits into two branches: `Error` and `Exception`
  - **`Error:`** internal errors and resource exhaustion inside the Java runtime system. Little you can do.
  - **`Exception:`** splits further into two branches.

Focus on the `Exception` branch

- Two branches of `Exception`
  - exceptions that derived from `RuntimeException`
    - examples: a bad cast, an out-of-array access
    - happens because errors exist in your program. Your fault.
  - those not in the type of `RuntimeException`
    - example: trying to open a malformed URL
    - program is good, other bad things happen. Not your fault.
- Checked exceptions vs. unchecked exceptions
  - ***Unchecked exceptions***: exceptions derived from the class `Error` or the class `RuntimeException`
  - ***Checked exceptions***: all other exceptions that are not unchecked exceptions
    - If they occur, they **must** be dealt with in some way.
    - The compiler will check whether you provide exception handlers for checked exceptions which may occur

Using `throw` to throw an exception

- Throw an exception under some bad situations

  E.g: a method named `readData` is reading a file whose header says it contains 700 characters, but it encounters the end of the file after 200 characters. You decide to throw an exception when this bad situation happens by using the `throw` statement

```
throw (new EOFException());
```

or,

```
EOFException e = new EOFException();
throw e;
```

the entire method will be

```
String readData(Scanner in) throws EOFException {
    . . .
    while(. . .) {
        if (!in.hasNext()) //EndOfFile encountered {
            if(n < len)
                throw (new EOFException());
        }
        . . .
    }
    return s; }
}
```

try/catch clause (1)

- Basic syntax of `try/catch` block

```
try {
    statements
} catch(exceptionType1 identifier1) {
    handler for type1
} catch(exceptionType2 identifier1) {
    handler for type2
} . . .
```

  - If no exception occurs during the execution of the statements in the **try** clause, it finishes successfully and all the **catch** clauses are skipped
  - If any of the code inside the **try** block throws an exception, either directly via a **throw** or indirectly:
    1. The program skips the remainder of the code in the **try** block
    2. The catch clauses are examined one by one, to see whether the type of the thrown exception is compatible with the type in **catch.**
    3. If an appropriate **catch** clause is found, the code inside its body gets executed and all the remaining **catch** clauses are skipped.
    4. If no such a **catch** clause is found, then the exception is thrown into an outer **try** that might have a **catch** clause to handle it

try/catch clause (2)

- ***Example***

```
public void read(String fileName) {
   try {
      InputStream in = new FileInputStream(fileName);
      int b;
```

//the `read()` method below is one which will throw an IOException

```
        while ((b = in.read()) != -1) {
        process input
      }
   } catch (IOException e) {
      exception.printStackTrace();
   }
}
```

Another choice for this situation is to do nothing but simply pass the exception on to the caller of the method:

```
public void read(String fileName) throws IOException {
   InputStream in = new FileInputStream(fileName);
   int b;
   while ((b = in.read()) != -1) {
process input
   }
}
```

finally clause (1)

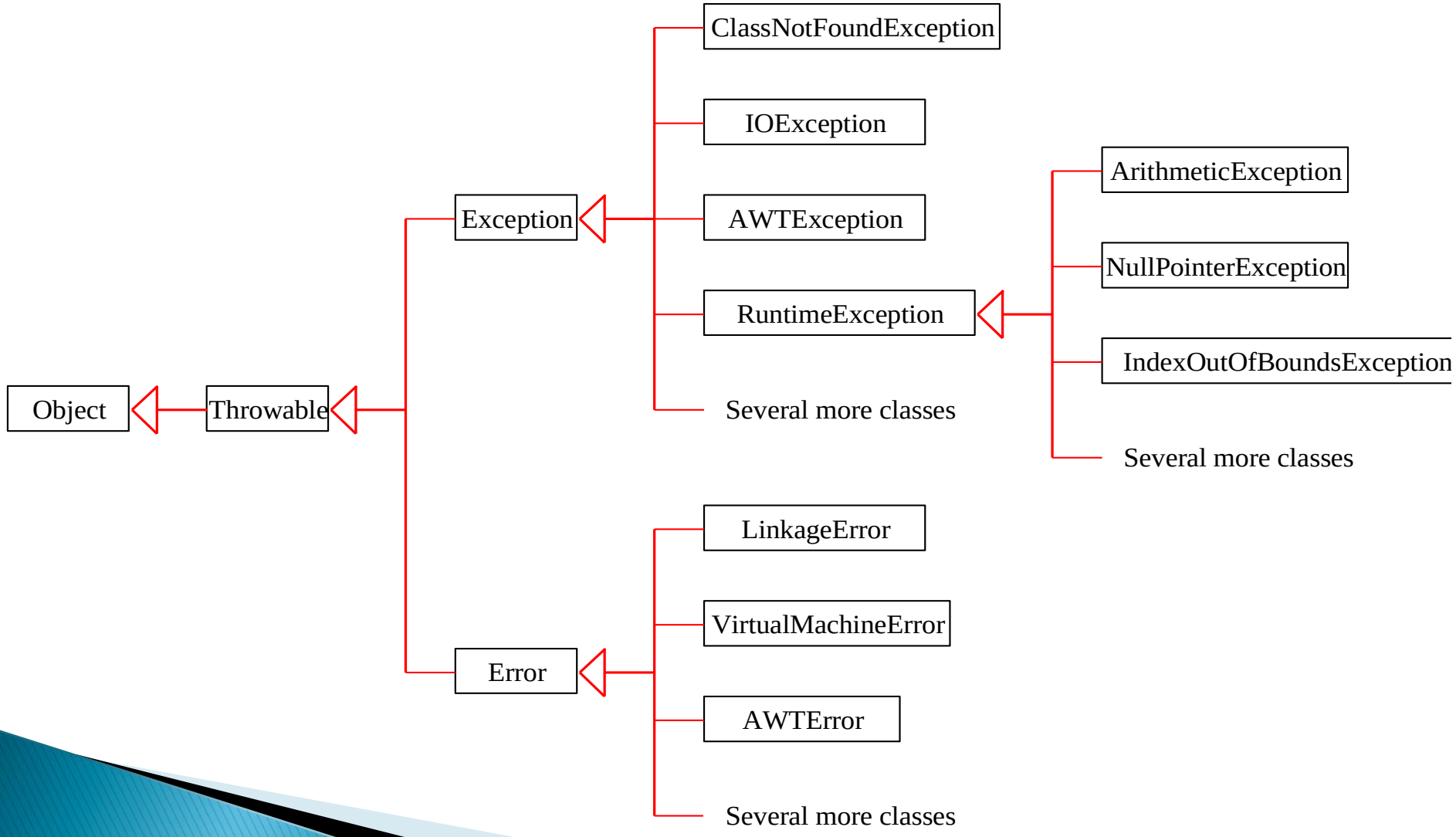
- You may want to do some actions whether or not an exception is thrown. **finally** clause does this for you:

```
Graphics g = image.getGraphics();
try {

     }
catch (IOException e) {

                              }
 finally {
     g.dispose();
           }
```

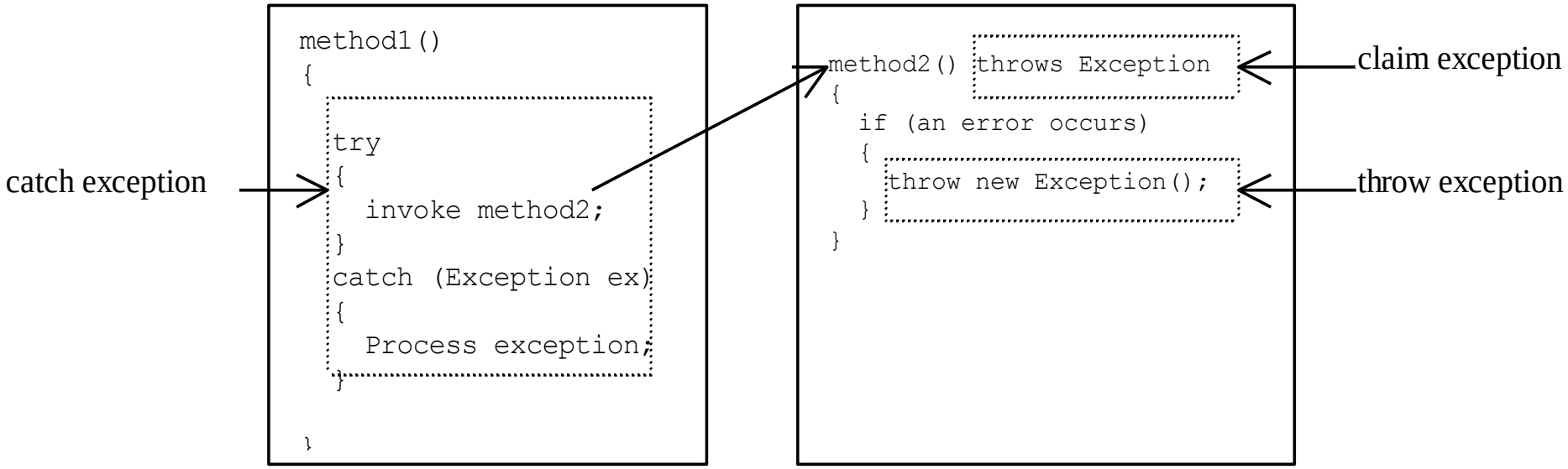
The `finally` Clause

```
try
{
   statements;
}
catch(TheException e)
{
   handling e;
}
finally
{
   finalStatements;
}
```

Exceptions and Exception Types



Claiming, Throwing, and Catching Exceptions



Claiming Exceptions

- ```
  public void myMethod()
    throws IOException
  ```

- Claiming multiple exceptions:

```
public void myMethod()
throws IOException, OtherException
```
```

Throwing Exceptions

- `throw new TheException();`
- ```
  TheException e = new TheException();
  throw e;
  ```
  ```

Example

```
public Rational divide(Rational r) throws
  Exception
{
  if (r.denom == 0)
  {
    throw new Exception("denominator
      cannot be zero");
  }

  long n = numer*r.denom;
  long d = denom*r.numer;
  return new Rational(n,d);
}
```

Catching Exceptions

```
try
{
  statements;
}
catch (Exception1 e)
{ handler for exception1 }
catch (Exception2 e)
{ handler for exception2 }

...

catch (ExceptionN e)
{   handler for exceptionN   }
```

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.