Exceptions and Assertions

Mārtiņš Leitass

Syntax Errors, Runtime Errors, and Logic Errors

- Errors can be classified into 3 categories:
  - Syntax errors.
    - They arise because the rules of the language have not been followed. They are detected by the compiler
  - Runtime errors
    - They occur while the program is running if the environment detects an operation that is impossible to carry out
  - Logic errors
    - They occur when a program doesn't perform the way it was intended to

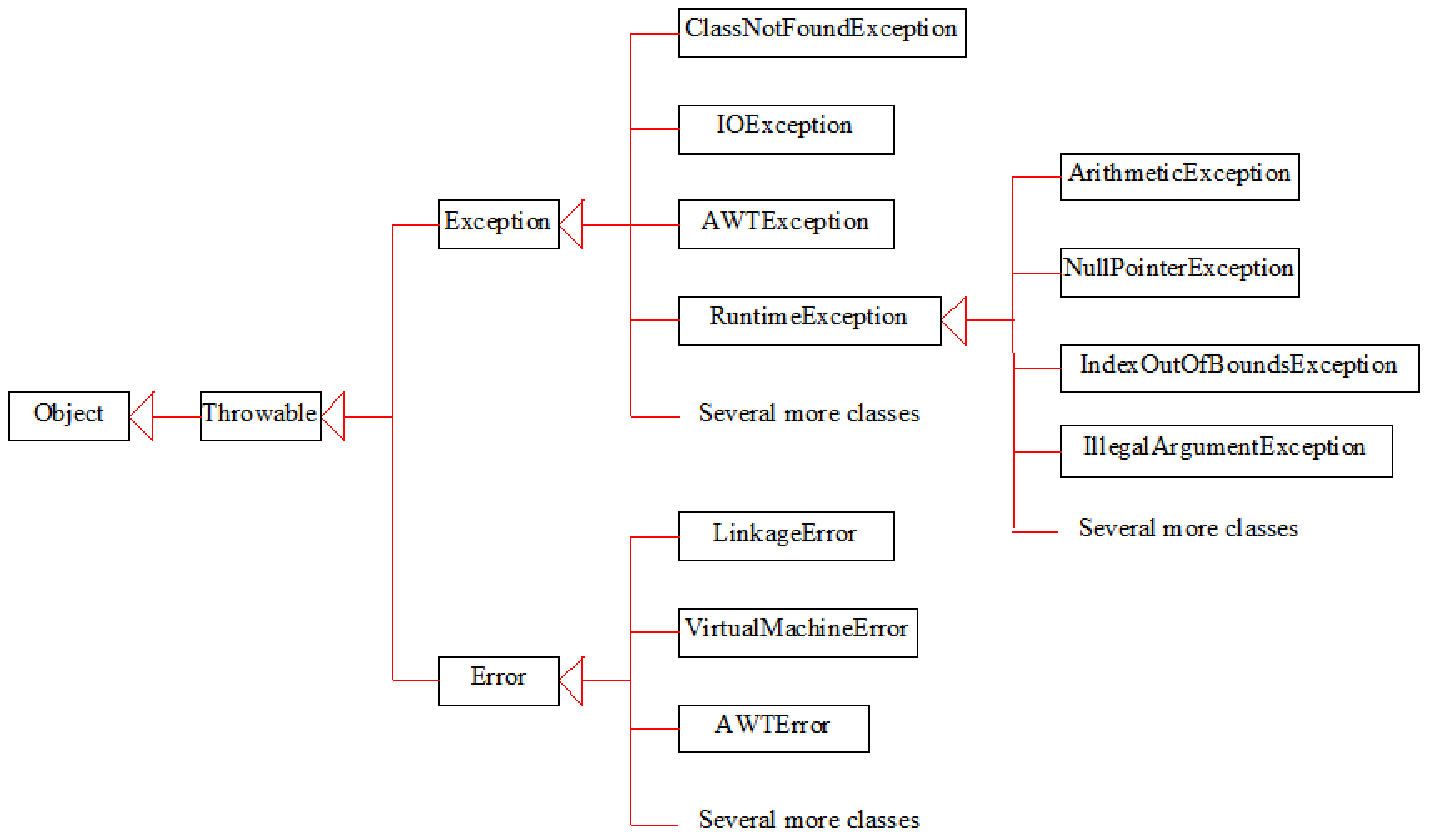
Exceptions and Assertions

- Exceptions handle unexpected situations – Illegal argument, network failure, or file not found
- Assertions document and test programming assumptions – This can never be negative here
- Assertion tests can be removed entirely from code at runtime, so the code is not slowed down at all.

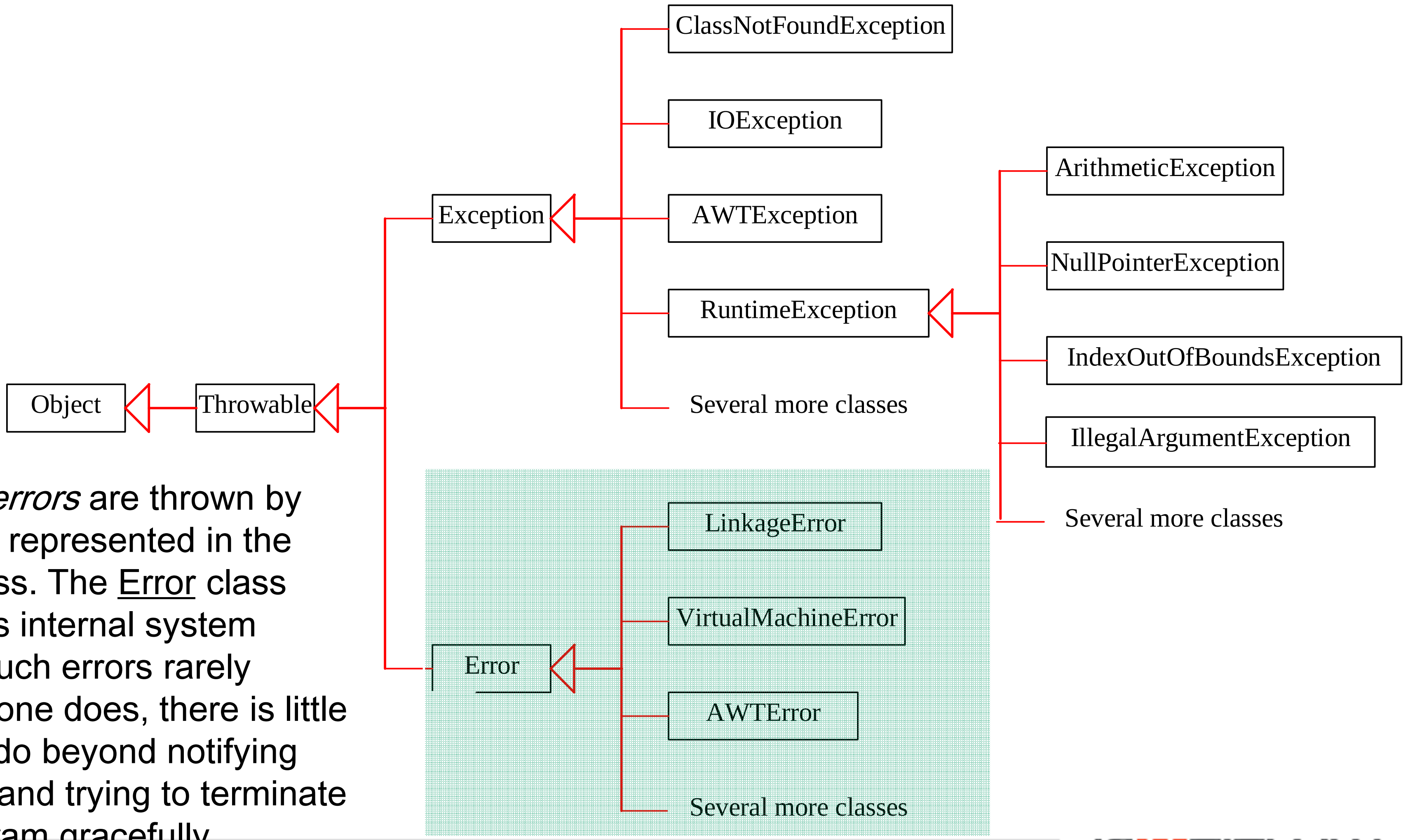
Exceptions

- Conditions that can readily occur in a correct program are *checked exceptions*. These are represented by the Exception class.
- Severe problems that normally are treated as fatal or situations that probably reflect program bugs are *unchecked exceptions*. Fatal situations are represented by the Error class. Probable bugs are represented by the RuntimeException class.
- The API documentation shows checked exceptions that can be thrown from a method.

Exception Classes



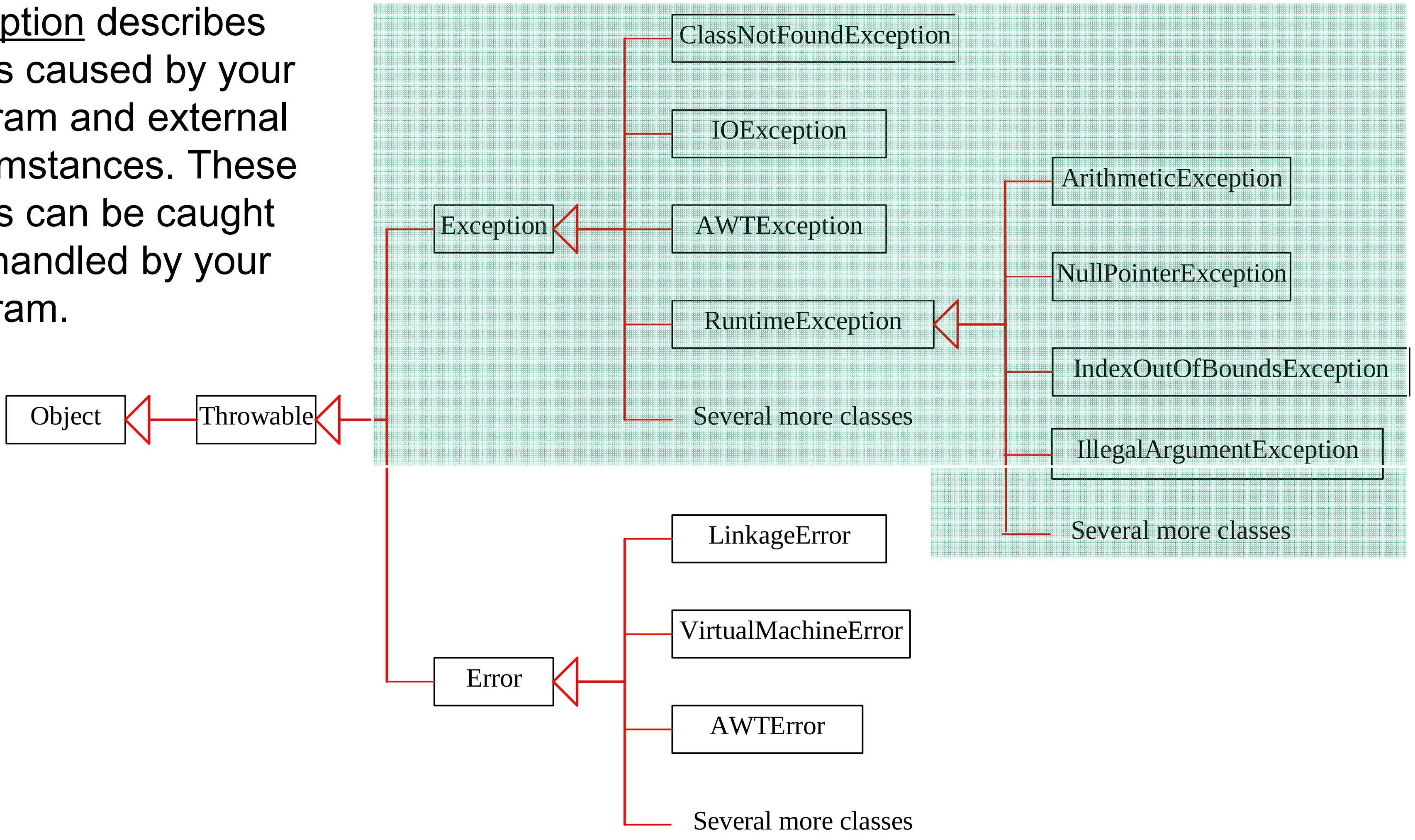
System Errors



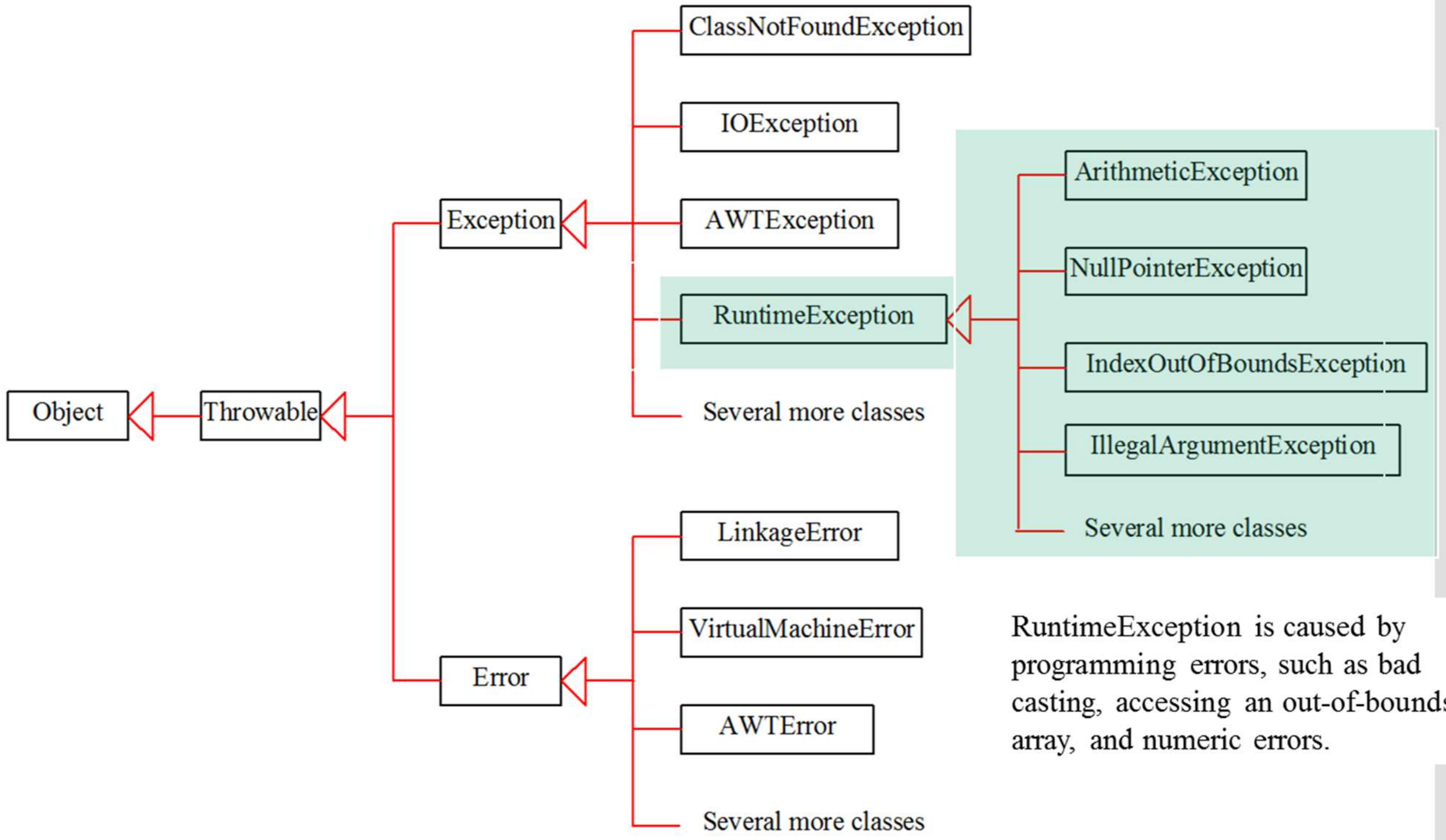
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Exception Example

```
1    public class AddArguments {
2      public static void main(String args[]) {
3        int sum = 0;
4        for ( String arg : args ) {
5          sum += Integer.parseInt(arg);
6        }
7        System.out.println("Sum = " + sum);
8      }
9    }
```

```
java AddArguments 1 2 3 4
Sum = 10
```

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

The try-catch Statement

```
public class AddArguments2 {
   public static void main(String args[]) {
      try {
         int sum = 0;
         for ( String arg : args ) {
            sum += Integer.parseInt(arg);
         }
         System.out.println("Sum = " + sum);
      } catch (NumberFormatException nfe) {
         System.err.println("One of the command-line "
                            + "arguments is not an integer.");
      }
   }
}
```

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

The try-catch Statement

```
public class AddArguments3 {
    public static void main(String args[]) {
        int sum = 0;
        for ( String arg : args ) {
            try {
                sum += Integer.parseInt(arg);
            } catch (NumberFormatException nfe) {
                System.err.println("[" + arg + "] is not an integer"
                                   + " and will not be included in the sum.");
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

The try-catch Statement

A try-catch statement can use multiple catch clauses:

```
try {
   // code that might throw one or more exceptions

} catch (MyException e1) {
   // code to execute if a MyException exception is thrown

} catch (MyOtherException e2) {
   // code to execute if a MyOtherException exception is thrown

} catch (Exception e3) {
   // code to execute if any other exception is thrown
}
```

Call Stack Mechanism

- If an exception is not handled in the current try-catch block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

The finally Clause

The finally clause defines a block of code that *always* executes.

```
try  {
    startFaucet();
    waterLawn();
}  catch  (BrokenPipeException  e)  {
    logProblem(e);
}  finally  {
    stopFaucet();
}
```

The finally block

- Is declared with the *finally* keyword
- it's statements are executed after all other *try-catch* processing is complete
- the finally clause executes whether or not an exception is thrown or a break or continue are encountered
- The finally block is a key tool for preventing resource leaks.
- Note: If a **catch** clause invokes **System.exit()** the finally clause WILL NOT execute.

Common Exceptions

Class Name	Exception Condition Represented
`ArithmeticException`	An invalid arithmetic condition has arisen, such as an attempt to divide an integer value by zero
`IndexOutOfBoundsException`	You've attempted to use an index that is outside the bounds of the object it is applied to. This may be an array, a String object, or a Vector object
`NegativeArraySizeException`	You tried to define an array with a negative dimension
`NullPointerException`	You used an object variable containing null, when it should refer to an object for proper operation
`ArrayStoreException`	You've attempted to store an object in an array that isn't permitted for the array type
`ClassCastException`	You've tried to cast an object to an invalid type—the object isn't of the class specified, nor is it a subclass or a superclass of the class specified
`IllegalArgumentException`	You've passed an argument to a method that doesn't correspond with the parameter type
`SecurityException`	Your program has performed an illegal operation that is a security violation. This might be trying to read a file on the local machine from an applet
`IllegalMonitorStateException`	A thread has tried to wait on the monitor for an object that the thread doesn't own
`IllegalStateException`	You tried to call a method at a time when it was not legal to do so
`UnsupportedOperationException`	This is thrown if you request an operation to be carried out that is not supported

The Handle or Declare Rule

Use the handle or declare rule as follows:

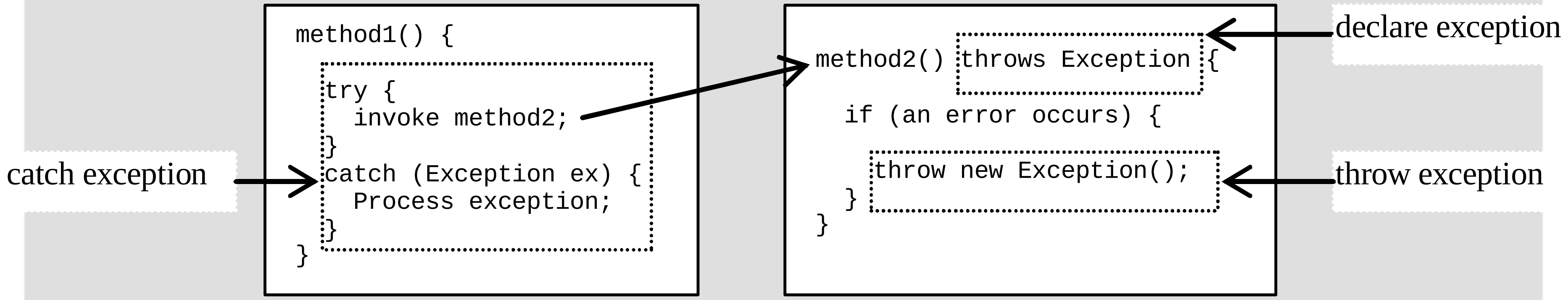
- Handle the exception by using the try-catch-finally block.
- Declare that the code causes an exception by using the throws clause.

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

Other Principles

- You do not need to declare runtime exceptions or errors.
- You can choose to handle runtime exceptions.

Declaring, Throwing, and Catching Exceptions



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Throwing Exceptions Example

```
/** Set a new radius */
 public void setRadius(double newRadius)
     throws IllegalArgumentException {
   if (newRadius >= 0)
     radius =  newRadius;
   else
     throw new IllegalArgumentException(
       "Radius cannot be negative");
 }
```

Method Overriding and Exceptions

The overriding method can throw:

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overriding method cannot throw:

- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method

Method Overriding and Exceptions

```
public class TestA {
   public void methodA() throws IOException {
      // do some file manipulation
   }
}
```

```
public class TestB1 extends TestA {
   public void methodA() throws EOFException {
      // do some file manipulation
   }
}
```

```
public class TestB2 extends TestA {
   public void methodA() throws Exception { //  WRONG
      // do some file manipulation
   }
}
```

Creating Your Own Exceptions

```
public class ServerTimedOutException  extends  Exception {
   private int port;

   public ServerTimedOutException(String message, int port) {
      super(message);
      this.port = port;
   }

   public int getPort() {
      return port;
   }
}
```

Use the getMessage method, inherited from the Exception class, to get the reason for which the exception was made.

Handling a User-Defined Exception

A method can throw a user-defined, checked exception:

```
public void connectMe(String serverName)
        throws ServerTimedOutException {
  boolean successful;
  int portToConnect = 80;

  successful = open(serverName, portToConnect);

  if ( ! successful ) {
    throw new ServerTimedOutException("Could not connect",
                                        portToConnect);
  }
}
```

Handling a User-Defined Exception

Another method can use a try-catchblock to capture user-defined exceptions:

```
public void findServer() {
        try {
            connectMe(defaultServer);
        } catch (ServerTimedOutException e) {
            System.out.println("Server timed out, trying alternative");
            try {
                connectMe(alternativeServer);
            } catch (ServerTimedOutException e1) {
                System.out.println("Error: " + e1.getMessage() +
                                                 " connecting to port " + e1.getPort());
            }
        }
    }
```

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program.
- An assertion contains a Boolean expression that should be true during program execution.
- Assertions can be used to assure program correctness and avoid logic errors.
- Assertions does not replace any kind of testing done by development team

lattelecom

Assertions

- Syntax of an assertion is:

assert ***<boolean_expression>*** **;**
assert ***<boolean_expression>*** **:** ***<detail_expression>*** **;**

- If *<boolean_expression>* evaluates false, then an AssertionError is thrown
- The second argument is converted to a string and used as descriptive text in the AssertionError message.

Executing Assertions Example

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

Recommended Uses of Assertions

- Use assertions to document and verify the assumptions and internal logic of a single method. Use assertions to reaffirm assumptions

Inappropriate Uses of Assertions

- Do not use assertions to check the parameters of a public method.
- Do not use methods in the assertion check that can cause side-effects
- Assertion should not be used to replace exception handling

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as if the check was never there.
- Assertion checks are disabled by default. Enable assertions with the following commands:

  ```
  java  -enableassertions  MyProgram
  ```

  or:

  ```
  java  -ea  MyProgram
  java  -ea:MyClass will enable only assertions for MyClass
  ```

- Assertion checking can be controlled on class, package, and package hierarchy bases, see: docs/guide/language/assert.html

Controlling Runtime Evaluation of Assertions

- Assertions can be selectively enabled or disabled at class level or package level
- The disable switch is –disableassertions or –da for short

```
java –ea:package1 –da:Class1 AssertionDemo
```

Assertions

- A good rule of thumb is that you should use an assertion for exceptional cases that you would like to forget about. An assertion is the quickest way to deal with, and forget, a condition or state that you don’t expect to have to deal with

Recommendations for exceptions

- **Use exceptions only for exceptional conditions.**
  That is, do not use exceptions for control flow, such as catching NoSuchElementException when calling Iterator.next() instead of first checking Iterator.hasNext().
- **Use checked exceptions for recoverable conditions and runtime exceptions for programming errors.**
  Runtime exceptions should be used only to indicate programming errors, such as precondition violations.
- **Avoid unnecessary use of checked exceptions**
  In other words, don't use checked exceptions for conditions from which the caller could not possibly recover, or for which the only foreseeable response would be for the program to exit.
- **Throw exceptions appropriate to the abstraction**
  In other words, exceptions thrown by a method should be defined at an abstraction level consistent with what the method does, not necessarily with the low-level details of how it is implemented. For example, a method that loads resources from files, databases, or JNDI should throw some sort of ResourceNotFound exception when it cannot find a resource (generally using exception chaining to preserve the underlying cause), rather than the lower-level IOException, SQLException, or NamingException.

Is the following code legal?

```
try {

} finally {

}
```

What exception types can be caught by the following handler?

```
catch (Exception e) {

}
```

What is wrong with using this type of exception handler?

Is there anything wrong with the following exception handler as written?

Will this code compile?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```

Testing in Java

- JUnit is most popular framework
- JUnit 4.x quick tutorial:
  - http://www.junit.org/node/477

NEW IN JAVA 7

P.S. For info only

- Multi catch
- Autoclosable

Multiple catch

- Before JDK 7

```
try {
    InputStream inStream = readStream(settingFile);
    Properties setting = parseFile(inStream);
} catch (IOException ex) {
    log.warn(“Can not access file”, settingFile);
} catch (FileNotFoundException ex) {
    log.warn(“Can not access file”, settingFile);
} catch (ParseException ex) {
    log.warn(“{} has incorrect format:{}”, settingFile,
                                            ex.getMessage());
}
```

Multiple catch

- With JDK 7

```
try {
    InputStream inStream = readStream(settingFile);
    Setting setting = parseFile(inStream);
} catch (IOException | FileNotFoundException ex) {
    log.warn(“Can not access file”, settingFile);
} catch (ParseException ex) {
    log.warn(“{} has incorrect format:{}”, settingFile,
                                        ex.getMessage());
}
```

Multiple catch

```
try {
  doWork(file);
} catch (IOException ex) {
  logger.log(ex);
}
catch (SQLException ex) {
  logger.log(ex);
}
```

```
try {
  doWork(file);
} catch (IOException|SQLException ex) {
  logger.log(ex);
}
```

Multiple catch

```
public  Setting readSettings(Strin settingFile)
       throws ParseException, IOException,
                                   FileNotFoundException {
    try {
        InputStream inStream = readStream(settingFile);
        Setting setting = parseFile(inStream);
    } catch (Throwable ex) {
        log.warn("Can not read settings", settingFile);
        throw ex;
    }
    ……………..
}
```

Try with resources

- Before JDK 7 (and still don’t forget about this approach):

```
private static String readConfiguration(String file) {
       BufferedReader reader = null;
       try {
           reader = new BufferedReader(new FileReader(file));
           String line = null;
           StringBuilder content = new StringBuilder(1000);
           while ((line = reader.readLine()) != null) {
               content.append(line);
           }
           return content.toString();
       } catch (IOException ex){
           throw new ConfigurationException("Can’t file:{}", file);
       } finally {
           if (reader != null)
               try {  reader.close();
               } catch (IOException ex) {}
```

Try with resources
JDK 7

```
private static String readConfigurationNew(String file) {
 try (BufferedReader reader = new BufferedReader(new FileReader(file));) {
                    String line = null;
                    StringBuilder content = new StringBuilder(1000);
                    while ((line = reader.readLine()) != null) {
                        content.append(line);
                    }
                    return content.toString();
          } catch (IOException ex){
              throw new ConfigurationException("Can't read file:{}", file);
          }
}
```

Try with resources

- A new interface AutoCloseable is introduced
- A new method addSuppressed(Exception) is added to Throwable
- Exceptions throwed from close method of AutoCloseable are suppressed in favor of exceptions throwed from try-catch block
- See JavaDoc of Autocloseable for more detail