Exceptions and Assertions

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/JavaSE

Exceptions

- The **Exception** class defines *mild* error conditions that your program encounters.
- The term *exception* is shorthand for the phrase "exceptional event." It can be defined as follows:

  Definition : An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

- The **Error** class defines *serious* error conditions.

Exceptions (Cont.)

- The exception handler chosen is said to *catch the exception*.

- Advantages :
  - Separating Error Handling Code from "Regular" Code
  - Propagating Errors Up the Call Stack
  - Grouping Error Types and Error Differentiation

Exception Categories
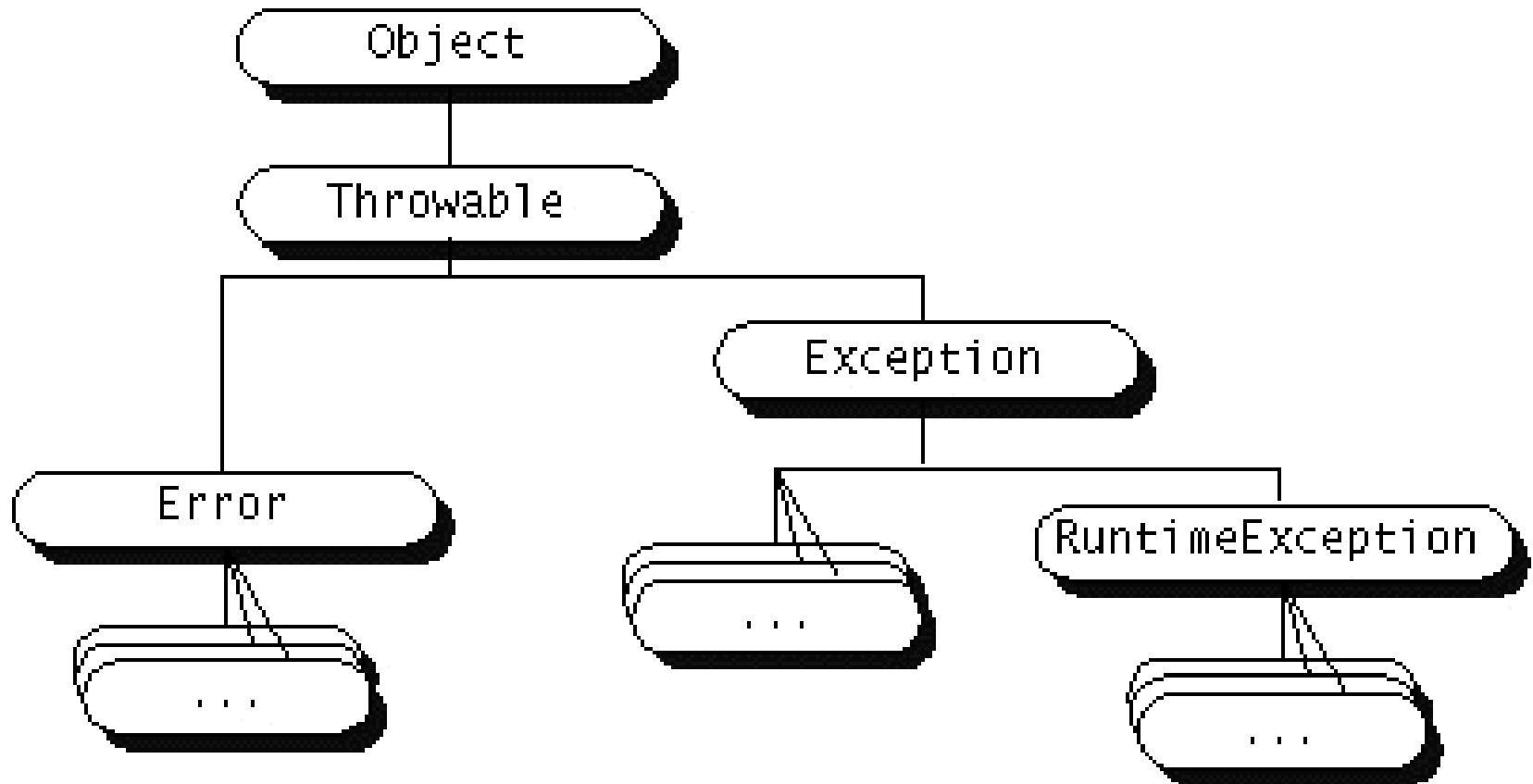
The Kind of Java's Exception

- Checked Exception
  - General Exception
  - Compiler checks if throw exceptions that a method declares.
  - Must use try-catch statement.
- Unchecked Exception
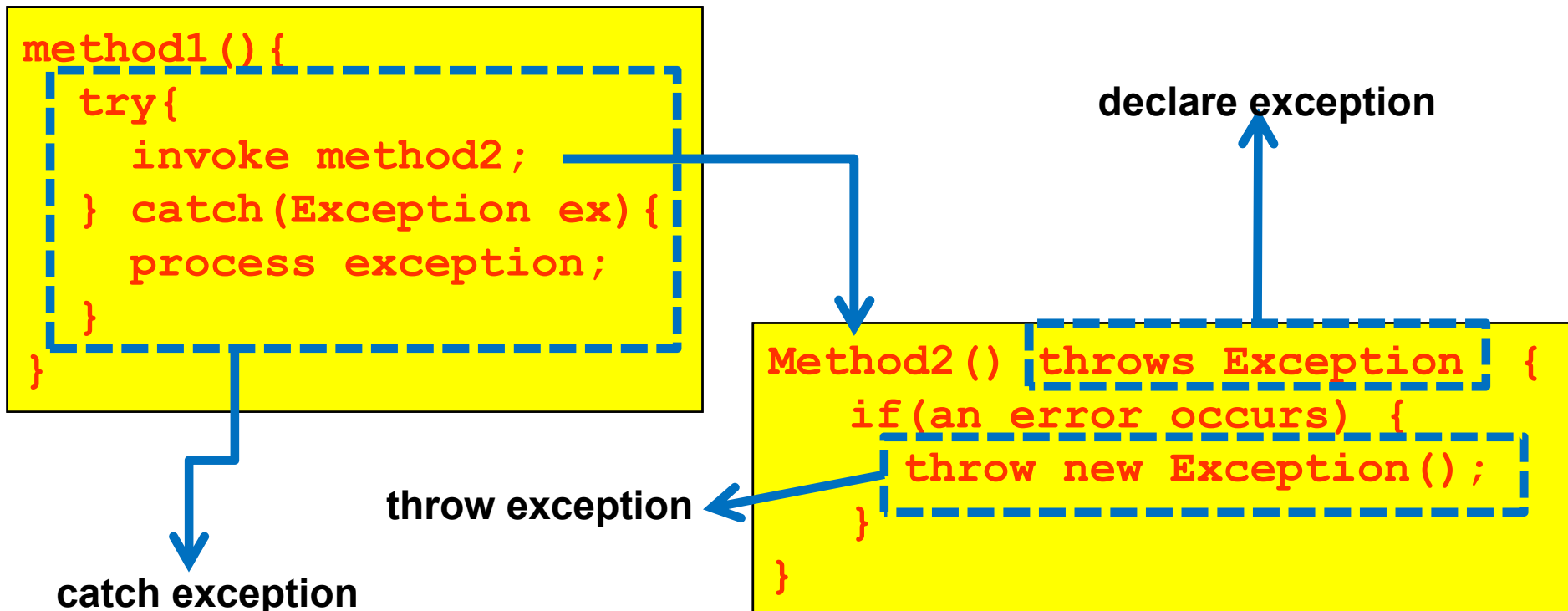 - `RuntimeException` or `Error` (or one of its subclasses).
  - Is not required to declare in its throws clause
  - Might be thrown during the execution of the method but not caught.

Throwable class

- Is the superclass of all errors and exceptions in the Java language.

- Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java **throw** statement.

- **String getMessage ()**

- **void printStackTrace ()**

- **String toString ()**

Understanding Exception Handling

- Java's exception-handling model is based on three operations: *declaring* an exception, *throwing* an exception, and *catching* an exception

```
method1(){
  try{
    invoke method2;
  } catch(Exception ex){
    process exception;
  }
}
```

declare exception

```
Method2() throws Exception {
    if(an error occurs) {
        throw new Exception();
    }
}
```

throw exception

catch exception

Catching Exceptions

- Java allows the program to catch and process exceptions.

- A `try` block contains the statements that *might* throw exceptions.

- If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.

- When a statement in a `try` block throws an exception, the rest of the statements in the `try` block are *skipped* and control is *transferred* to the `catch` block.

- The code that handles the exception is called the *exception handler*.

Catching Exceptions (Cont.)

- A try block begins with the keyword **try** followed by a block of statements in curly braces (**{}**).

- A catch block begins with the keyword **catch** followed by an exception parameter in parentheses and a block of statements for handling the exception in curly braces.

```
try {
    statements; // Statements that may
                // throw exceptions
}catch (Exception ex) {
    handler for exception;
}
```

Using `try` and `catch` Blocks

- Object-oriented solution to error handling
  - Put the normal code in a `try` block
  - Handle the exceptions in a separate `catch` block

```
int su = 5, num = 0;
try {
      System.out.println("su / num = " + su/num);
}catch (ArithmeticException e){
      System.out.println("Invalid Value");
}
```

Program logic

Error handling

The **Exception** Handling Process

1. **Exception** is encountered resulting in an exception object being created.
2. A new exception object is thrown.
3. The runtime system looks for code to handle the exception.
   1. If no handler is found, the runtime environment traverses *the call stack* (the ordered list of methods) in reverse looking for an exception handler.
   2. If the exception is not handled, the program exits and a *stack trace* is automatically output.
4. The runtime system hands the exception object off to an exception handler to handle (**catch**) the exception.

Multiple `catch` Blocks

- Each `catch` block catches one class of exception.
- A `try` block can have one general `catch` block.
- A `try` block is not allowed to catch a class that is derived from a class caught in an earlier `catch` block.

```
int su = 5, num = 0;
try {
        System.out.println("su / num = " + su / num);
}catch (NegativeArraySizeException e) {…
}catch (ArrayIndexOutOfBoundsException e) {…
}catch (ArithmeticException e){…
}
```

Multiple `catch` Blocks (Cont.)

- The order in which exceptions are specified in `catch` blocks is important.
- A compilation error will result if a `catch` block for a superclass type appears before a `catch` block for a subclass type.

```
try {
    ….
}catch (Exception ex) {
    ….
}catch(RuntimeException ex){
    ….
}
```
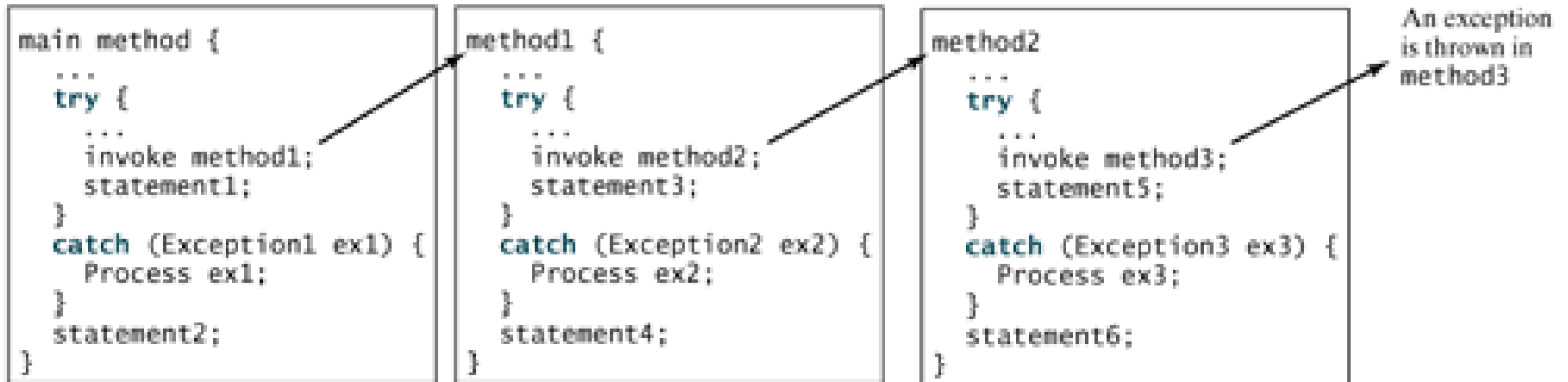
```
try {
    ….
}catch(RuntimeException ex){
    ….
}catch(Exception ex){
    ….
}
```

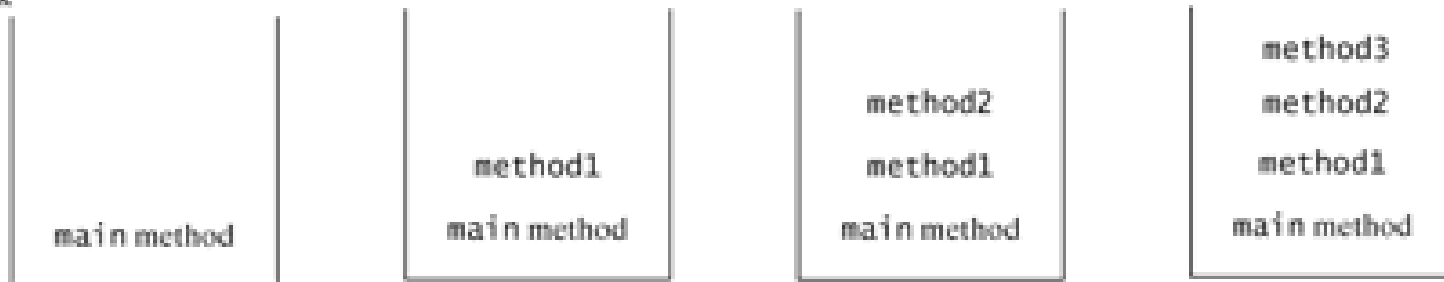**Wrong Order**   **Correct Order**

Calling Stack

- Exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

- Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block.

  - If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.

  - If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.

Calling Stack (Cont.)

```
main method {                  method1 {                     method2
   ...                            ...                            ...
   try {                          try {                          try {
     ...                            ...                            ...
     invoke method1;                invoke method2;                invoke method3;
     statement1;                    statement3;                    statement5;
   }                              }                              }
   catch (Exception1 ex1) {      catch (Exception2 ex2) {      catch (Exception3 ex3) {
     Process ex1;                   Process ex2;                   Process ex3;
   }                              }                              }
   statement2;                    statement4;                    statement6;
}                              }                              }
```

An exception
is thrown in
method3

Call Stack

```
                                                                              method3
                              method2         method2
            method1           method1         method1
main method main method       main method     main method
```

Throwing Exceptions

- Throw an appropriate exception.
- Give the exception a meaningful message.

```
throw expression ;
```

```
if (minute < 0 || minute >= 60) {
  throw new InvalidTimeException (minute +
                            " is not a valid minute");
  // !! Not reached !!
}
```

Declaring Exceptions

- Every method must state the types of checked exceptions it might throw so that the caller of the method is informed of the exception.

- While, Java does not require that you declare **Error** and **RuntimeException** (unchecked exceptions) explicitly in the method.

- The **throws** keyword indicates that myMethod might throw an **IOException**.

```
public void myMethod() throws IOException {
    .....
}
```

Declaring Exceptions (Cont.)

- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after **throws**:

```
public void myMethod() throws Exception1 [
, Exception2, ..., ExceptionN ]
```

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

Printing Information About Exceptions

- The **getMessage()** method
  - Returns a detailed message string about the exception
- The **toString()** method
  - Returns a detailed message string about the exception.
  - Including its class name

```java
try {
    new FileReader("file.dat");
} catch (FileNotFoundException  ex) {
    System.err.println(ex.getMessage( ));
}
```

Printing Information About Exceptions (Cont.)

- ## The `printStackTrace()` method

  - Returns a detailed message string about the exception,

  - Including its class name and

  - A stack trace from where the error was caught, all the way back to where it was thrown.

```
java.io.FileNotFoundException: file.js
(The system cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.(init)(FileInputSteam.java: 106)
at java.io.FileInputStream.(init)(FileInputSteam.java:66)
at java.io.FileReader(init)(FileReader.java:41)
at EHExample.openFile(EHExample.java:24)
at EHExample.main(EHExample.java:15)
```

The `finally` Statement

- The `finally` statement defines a block of code that *always* executes, regardless of whether an exception was caught.
- The `catch` block may be omitted when the `finally` clause is used.

```
try {
     conn = DriverManager.getConnection (…);
    ...
}catch (SQLException e){…
}finally {
    if (conn != null) conn.close();
}
```

Creating Your Own Exception

- To define a checked exception, the new exception class *must extend* class `Exception`, directly or indirectly.

- To define an unchecked exception, the new exception class *must extend* class `RuntimeException`, directly or indirectly.

- To define an unchecked error, the new error class *must extend* class `Error`.

- User-defined exceptions should have at least *two constructors*: a constructor that does not accept any arguments and a constructor that does.

```java
public class SimpleException extends Exception {
    public SimpleException(){}
    public SimpleException(String msg) {
        super(msg);
    }
}
```

Assertions

- The logic in your code leads to some logical condition that should always be *true*.

- Are **Boolean** expressions used to check whether code behaves as expected while running in debug mode

```
// num value should be between 1 and 18.
assert (num >= 1 && num <= 18);
```

- Help identify bugs more easily, including identifying unexpected values.

- Are designed to validate assumptions that should al ways be *true*.

Assertions (Cont.)

- While running in debug mode, if the assertion evaluates to *false*, a `java.lang.AssertionError` is thrown and the program exits;

- Otherwise, nothing happens.

- Assertions need to be explicitly enabled.

- Syntax :
  - `assert logical_expression;`
  - `assert logical_expression : message;`

Assertions – How It Works

- **`assert`** is a keyword

- logical_expression is any expression that results in a value of **`true`** or **`false`**.

- If logical_expression evaluates to **`true`**, then the program continues normally.

- If logical_expression evaluates to **`false`**, the program will be terminated with an error message starting with:

  `java.lang.AssertionError`

Assertion – Sample Code

```java
1  public class AssertionDemo {
2      public static void main(String[] args) {
3          int i, sum = 0;
4          for( i = 0 ; i < 10 ; i++){
5              sum += i;
6          }
7          assert i == 10;
8          assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
9      }
10 }
```

Assertion – Sample Code

- If i is not 10, an **AssertionError** is thrown.

- Asserts that sum > 10 and sum < 5 * 10. If false, an **AssertionError** with the message "sum is " + sum is thrown.

- Suppose

  - Types i < 100 instead of i < 10 by mistake in line 4, the following **AssertionError** would be thrown:

    ```
    Exception in thread "main" java.lang.AssertionError
        at AssertionDemo.main(AssertionDemo.java:7)
    ```

  - Types sum += 1 instead of sum += i by mistake in line 5, the following **AssertionError** would be thrown:

    ```
    Exception in thread "main" java.lang.AssertionError: sum is 10
        at AssertionDemo.main(AssertionDemo.java:8)
    ```

How to Compile and Interpretation

- Compile

  `javac -source 1.5 AssertionTest.java`

- By default, assertions are disabled at runtime.

- To enable them, use the switch

  `-enableassertions`, or **`-ea`** for short,
  as follows:

  `java -ea AssertionTest`

How to Compile and Interpretation (Cont.)

- Assertions can be selectively enabled or disabled at the class level or the package level.
- The disable switch is **−disableassertions**, or **−da** for short.

 ```
 java −ea:package1 −da:Class1 AssertionDemo
 ```
- **java -ea** → Enable Assertion
- **java −da** → Disable Assertion (default)
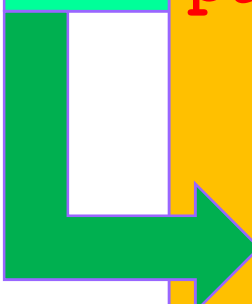- **java −ea:package.name**
- **java −ea:className**

Exception vs Assertion

- Assertion should not be used to replace exception handling.

- Exception handling deals with unusual circumstances during program execution.

- Assertions are intended to ensure the correctness of the program.

- Exception handling addresses robustness, whereas assertion addresses correctness.

- Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks.

- Assertions are checked at runtime and can be turned on or off at startup time.

Exception vs Assertion (Cont.)

- Do not use assertions for argument checking in public methods.

- The contract must always be obeyed whether assertions are enabled or disabled.

```java
public void setRadius(double newRadius) {
    assert newRadius >= 0;
    radius = newRadius;
}
```

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

Exception vs Assertion (Cont.)

```
switch (month) {
    case 1  :    .... ;    break;
    case 2  :    .... ;    break;
        ....
        ....
    case 12 :    .... ;    break;
    default : assert  false : "Invalid month : " + month;
}
```

```
 if (numberOfDollars > 1 ) {
      ….
} else if (numberOfDollars == 1) {
      ….
} else
      assert false : numberOfDollars ;
```