Exception Handling

Topics:

including

- (Types of) Exceptions: Checked *versus* Unchecked
- **`try/catch`** & **`finally`** blocks
- **`throw`** *versus* **`throws`**
- Declaring Exceptions // Catching Multiple Exceptions
- Assertions

Chapter 10 – "Big Java" book
Chapter 11 – "Head First Java" book
Chapter 17 – "Introduction to Java Programming" book
Chapter 5 – "Java in a Nutshell" book

Errors in a Java Program (1/2)

- Some causes of error situations:
  - Incorrect implementation, e.g. when a program does not meet the specification.
  - Inappropriate object request, e.g. when trying to access an invalid index.
  - Inconsistent or inappropriate object state, e.g. following a class extension.
- Errors are not always due to programmer error:
  - Errors often arise from the environment, e.g. an incorrect URL entered or a network interruption.
  - File processing is particularly error-prone, e.g. due to missing files and lack of appropriate permissions.

Errors in a Java Program (2/2)

- Many types of errors can occur when running a program, some of which are difficult to predict or prevent:
  - Examples: incorrect input; a host server that is unavailable …
- More common errors are programming errors (or errors in the program's logic). Examples:
  - Trying to access an array out of bounds → this throws an **`ArrayIndexOutOfBoundsException`** runtime error.
  - Attempting to divide by zero → this throws an **`ArithmeticException`** runtime error.
- When Java detects an error at *runtime*, an *exception* is thrown.
- Exception: an object that signals to the calling code, the occurrence of an unusual condition.

Could we use **`if`** statements to solve problem?

Run-Time Error Handling

- Run-time programming errors are the most difficult to deal with.
- It's difficult to predict all the possible error states that a program can create while designing and implementing the program.
  - Possible result: a program crash, and subsequent user frustration!
- There's *no support* given at *programming language level* for catching and managing errors in most languages.

Run-Time Errors in Java

- Java *forces* programmers to either *catch* run-time errors or else *declare* that they are not catching them, through run-time constructs:

  `try` `catch` `finally` — to *detect* & *handle* exceptions

  `throws` `throw` — to *declare* & *throw* exceptions

  – This is based on knowing (by looking for the keyword `throws`) that the method you're calling might generate an exception.

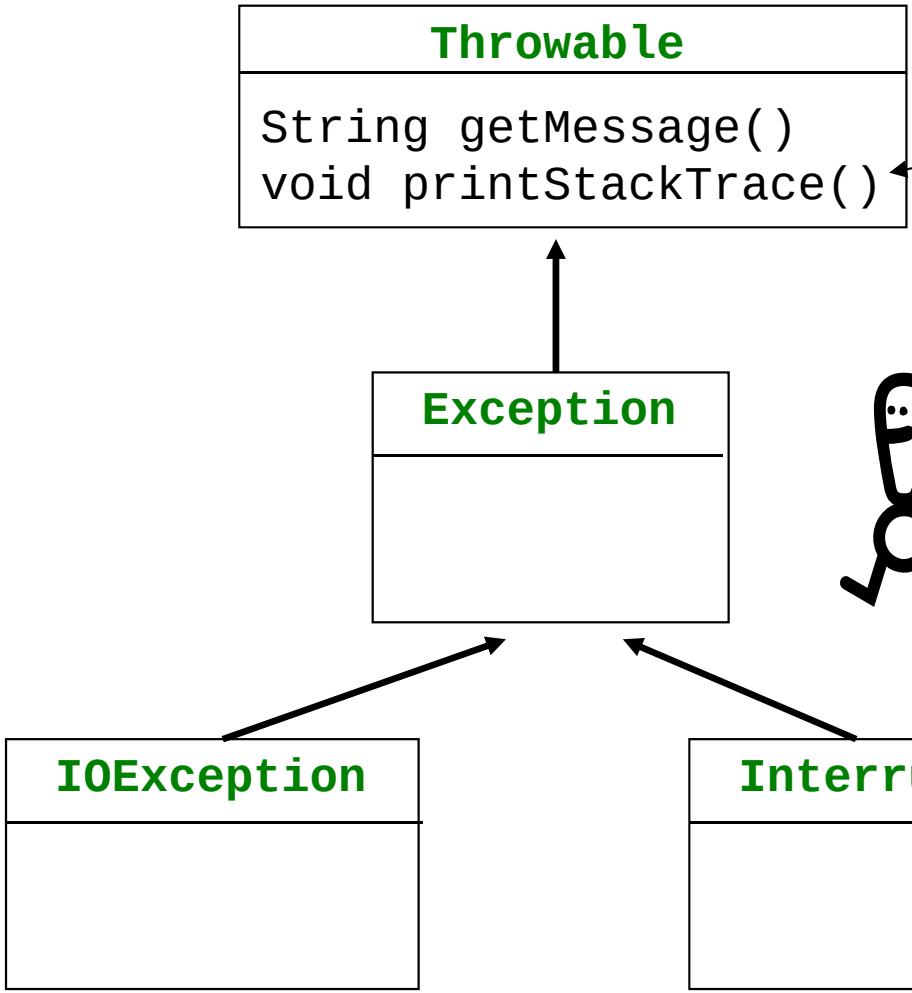
- Using exceptions enables potential *run-time* problems to be noticed at *compile time*: *much* more effective than exhaustive testing!

Trying & Catching Exceptions

- Programmers declare all possible errors or unusual circumstances that may be caused by their classes: these are *exceptions*.
- Any code that wishes to make use of these classes must either *catch* these exceptions, or else explicitly declare that they are *not* caught.
  - If an exception is caught, then control is transferred to a special block of code where it is *handled*.
- Exceptions are *objects*, subclasses of **`java.lang.Exception`** class.

A method ***catches*** what another method ***throws***.
Exceptions are always thrown back to the caller code.

The `Exception` Class Hierarchy



can be used when you *don't know how to recover* from an error

Using polymorphism again: an object of type **`Exception`** can be an instance of any subclass of **`Exception`**.

Example: Catching File I/O Errors

- What happens when you try to read in data from a file?
  1. Open file.
  2. Allocate memory for file.
  3. Read file into memory.
  4. Close file.
- **Problems** can happen at any stage. What if:
  - the file isn't there?
  - the file can't be opened?
  - there isn't enough memory left to read in the file?
- Using a **`try/catch`** block:

```
try {
  readFromFile("foo");
} // end try
catch (Exception e) {
  // handle error
  System.err.println("Read file exception:" + e);
} // end catch
```

What Happens in a `try/catch` Block

- When a program is run, the JVM will attempt to execute each statement in the **`try`** block in turn.
- If any statement ***throws an exception***, then either:
  - the **`catch`** block corresponding to this exception will be executed; *OR*
  - the method in which this code lies will itself throw the exception.
- A **`try/catch`** section can also have a **`finally`** section, usually to *tidy up* afterwards (e.g. to close files).

Syntax for `try/catch/finally` Blocks

```
try {
   // code that can throw exceptions E1, ..., En
   // ...
}
catch (E1 e1) {
   // code to handle exception E1
   // ...
}
// ...
catch (En en) {
   // code to handle exception En
   // ...
}
finally {
   // code to tidy up: close files, etc
}
```

protect one or more statements here

report and recover from the exception here

perform any actions here that are common, regardless of whether or not an exception is thrown

The **`finally`** block is optional.

Content of a `try` Section

- A **`try`** section will "bomb out" if it encounters an exception.
  - This means that **local variables** can be left without properly initialised values:

```
int foo;
try {
  // ...
  foo = getResults();
}
catch (Exception e) { // ... }
int bar = foo; // HERE!
```

  - Partial solution:

```
int foo = 0;
try {
  // ...
  foo = getResults();
  int bar = foo;
  // ...
}
catch (Exception e) { // ... }
// ...
```

But this means putting *everything* in the **`try`** section; this is known as **`try`** *creep*.

Example: Throwing & Catching Exceptions (1/2)

- When writing code, we can *throw* exceptions, and thus force any clients that use it to *catch* these exceptions.
  - Example:

declares it *throws* exception of type **`Exception`**

```
public class RiskyClass {
  public void checkFileName(String s) throws Exception {
    if (s.equals("/etc/passwd"))
      throw new Exception("bad filename");
  }
}
```

risky code

causes exception to be *thrown*

client code trying to use *risky* code

```
public class TestExceptions {
  public static void main(String[] args) {
    RiskyClass rc = new RiskyClass();
    for (int i = 0; i < args.length; i++) {
      rc.checkFileName(args[i]);
    } // end for
  }   // end main()
}
```

Example: Throwing & Catching Exceptions 2/2

client code now *handling exceptions*

```
public class TestExceptions{
  public static void main(String[] args) {
   RiskyClass rc = new RiskyClass();
   for(int i = 0; i < args.length; i++) {
     try {
       rc.checkFileName(args[i]);
     } // end try
     catch (Exception e) {
       System.err.println(""+ e + " at "+ i);
     } // end catch
   }   // end for
  } // end main()
}
```

```
>javac TestExceptions.java

>java TestExceptions myfile

>java TestExceptions myfile /etc/passwd
java.lang.Exception: bad filename at 1
```

Ducking: Alternative to Catching Exceptions

- Sometimes, you *don't* want to catch exceptions, but want client code to *handle* them (*aka* ducking or hiding).
  - In this case, you should declare that your method **`throws`** them:

```
/**
  @throws Exception textWithReasonForException
 */
public void checkNames(String[] args) throws Exception {
  for (int i = 0; i < args.length; i++) {
    checkFileName(args[i]);
  } // end for
}   // end method checkNames()
```

Javadoc documentation syntax to indicate that method *throws* an exception: `@throws ExceptionType reason`

… and things for you to try out!

Exceptions: Warnings, When to Use & When to Throw

- Warnings:
  - Programs containing exception handling are easier to read/modify but need more work when implementing because,
    - Exception objects need instantiating.
    - Errors need to be propagated to the calling methods.
- When to,
  - *throw exceptions*: when you want the caller of a method to process an error;
  - *declare exceptions*: when you need to resolve unexpected error conditions.

Creating Exception Classes

- Java programmers can create their own exception classes.
  - **User exception classes** are like any other class, but they must extend the **`Exception`** class.
  - **Example** of typical syntax:

```
public class MyException extends Exception {
  public MyException()  {
    super(); // call constructor of parent Exception
    // other appropriate code
  }
  public MyException(String s)  {
    super(); // call constructor of parent Exception
    // other appropriate code
  }
}
```

Example using Exceptions (1/4)

- Problem:
  - Create **`Date`** object, representing a date in a Gregorian calendar.
  - Ensure that invalid dates (i.e. dates that don't exist in the Gregorian calendar) are dealt with by *throwing* our own exception **`InvalidDateException`**.

```
/**
* InvalidDateException: Example of creating an exception class.
*
* @author R J Mondragon
*/
public class InvalidDateException extends Exception {
  public InvalidDateException() {
    // here we create the exception
    super("Invalid date: please try again ...");
  }
}
```

Example using Exceptions (2/4)

Method **`setDate()`** can be improved. How?

```
/**
* MyDate: This class stores and
* manipulates dates on the Gregorian
* calendar. Throws InvalidDateException:
* date has the wrong format.
*
* @author R.J.Mondragon
*/
public class MyDate {
  // instance variables
  private int year, month, day;
  public MyDate() { // Default date
    year = 1900;
    month = 1;
    day = 1;
  }

  public MyDate(int day, int month,
                int year)
    throws InvalidDateException {
      setDate(day, month, year);
    }
```

```
  public void setDate(int day, int month,
                      int year)
    throws InvalidDateException {

    if (year < 0) {
      throw new InvalidDateException();
    }
    else { this.year = year; }

    if ((month < 0) || (month > 12)) {
      throw new InvalidDateException();
    }
    else { this.month = month; }

    if ((day < 0) || (day > 31)) {
      throw new InvalidDateException();
    }
    else { this.day = day; }
  }
}
```

Example using Exceptions (3/4)

```
/**
 * TestMyDate: Test MyDate class.
 *
 * @author RJ Mondragon
 */
public class TestMyDate {
  public static void main(String[] args) throws InvalidDateException {
    MyDate d = new MyDate(10, 11, -1980);
  }
}
```

Main program.

Our code signals that it can *throw* an exception but doesn't *catch* it.

- **Output** is ...

```
> java TestMyDate
Exception in thread "main" InvalidDateException: Invalid date:
please try again ...
at MyDate.setDate(MyDate.java:14)
at MyDate.<init>(MyDate.java:10)
at TestMyDate.main(TestMyDate.java:3)
```

  - Java throws the exception because we input a negative year value.

Example using Exceptions (4/4)

- **Catching** (i.e. *handling*) the exception:

```
/**
 * NewTestMyDate: This class creates a Date object with an
 * invalid date, and catches the error in an exception handler.
 *
 * @author R J Mondragon
 */
public class NewTestMyDate{
  public static void main(String[] args) {
    try { // first try
      MyDate d = new MyDate(10, 11, -1980);
    }
    catch (InvalidDateException e) {
      System.err.println("The exception is:\n" + e.getMessage());
    }
    finally { System.out.println("finally always executes ..\n"); }
  }
}
```

Output is ...

```
> java NewTestMyDate
The exception is:
Invalid date: please try again ...
finally always executes ...
```

… and things for you to try out!

Understanding Stack Trace Messages

- If a *program fails to catch an exception*, the JVM interpreter prints information about the exception, and the location where it occurred.
- Example & Interpretation:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at PrintCalendar.daysInMonth(Printcalendar.java:121)
at PrintCalendar.main(Printcalendar.java:42)
```

The exception named **`ArithmeticException`** occurred. This exception belongs to the package **`java.lang`**.

It occurred in line **`121`** of file **`PrintCalendar.java`**; the method being executed at the time was **`daysInMonth()`**.

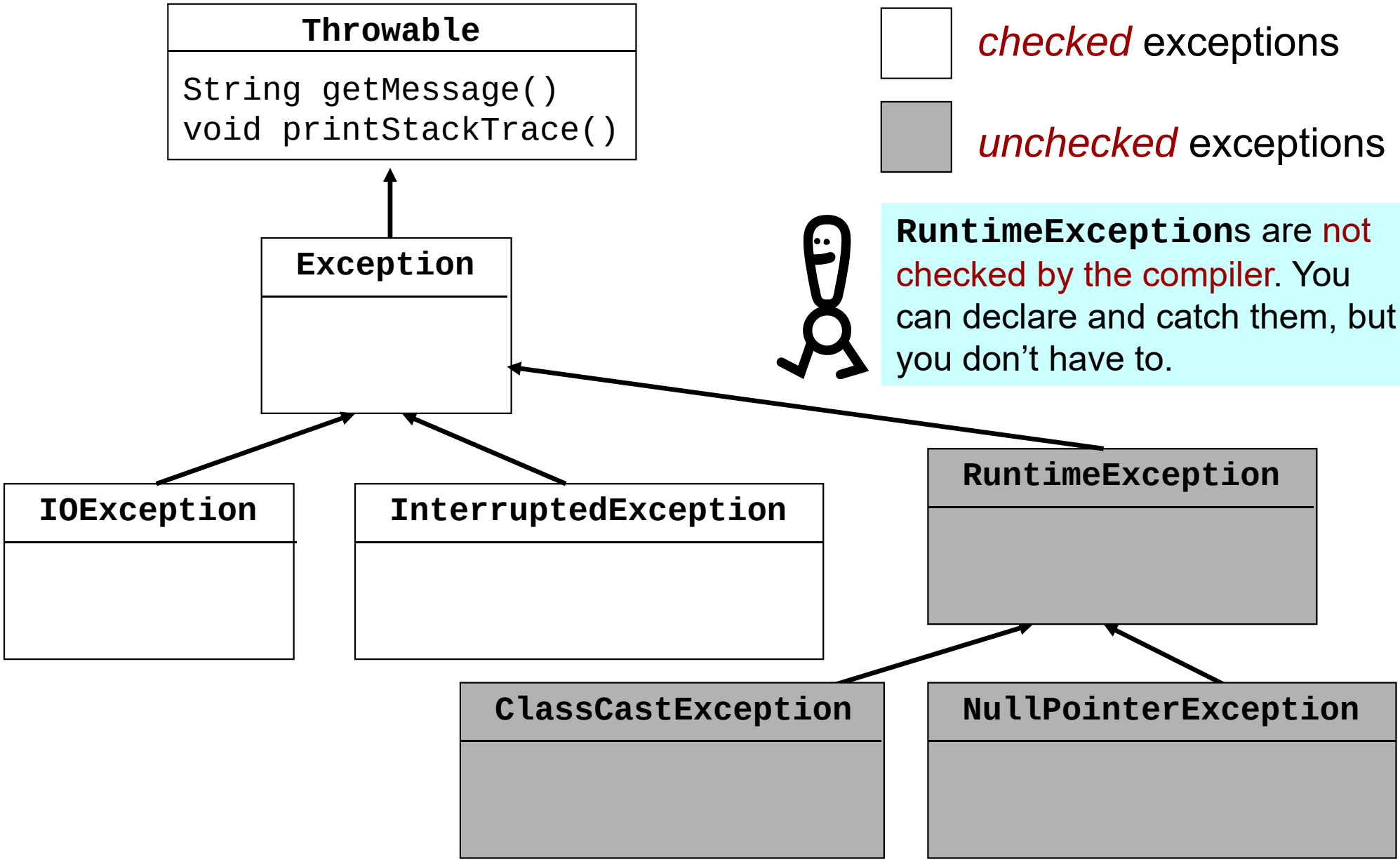
The **`daysInMonth()`** method had been called on line **`42`** of file **`PrintCalendar.java`** by the **`main()`** method.

Exercise 1

- Consider the code fragment below. Assuming that **statement2** causes (or throws) an exception, answer the following questions:
  - Will **statement3** be executed?
  - If the exception is not caught, will **statement4** be executed?
  - If the exception is caught in one of the **catch** clauses, will **statement4** be executed?

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) { }
catch (Exception2 ex2) { }
statement4;
```

Checked *versus* Unchecked Exceptions



Exception Categories & Run-Time Exceptions

- Checked exceptions:
  - Subclasses of `Exception`.
  - Used for anticipated failures.
  - Where recovery may be possible.
- Unchecked exceptions:
  - Subclasses of `RuntimeException`.
  - Used for unanticipated failures.
  - Where recovery is unlikely.

- Some exceptions thrown by Java class libraries are called *run-time exceptions*.
- Java does not force client code to catch run-time exceptions (also called *unchecked exceptions*), because:
  - Run-time exceptions can occur so frequently that the cost of checking by the compiler would be very big.
  - You *can* catch them if you believe there is ever likely to be a problem.
  - Ideally, you should instead check input pre-conditions first! (e.g. what is the effect of mutator methods).

Examples: Run-Time (RT) Exceptions

- (Some of the) *RT Exceptions* not automatically checked by the compiler:

Exception Class	Meaning
`NullPointerException`	Accessing an object (reference) variable.
`ArrayStoreException`	Attempting to store the wrong type of object into an array of objects.
`IndexOutOfBoundsException`	Using an index of some sort, out of bounds
`NegativeArraySizeException`	Trying to create a negative-size array.
`ArithmeticException`	Example: attempting a division by zero.

All exception types can be found on the Java API.

- Using the Java class libraries involves learning about the exceptions thrown and catching them!
- The exceptions a method throws are part of its *interface*.
- Whenever possible, make use of existing exception types, e.g. `IllegalArgumentException`.

Exercise 2

- What is the output of this program?

Assume class **ScaryException** was defined somewhere else.

```
public class TestExceptions {
  public static void main(String[] args) {
    String test = "no";
    TestExceptions tex = new TestExceptions();
    try {
      System.out.println("start try");
      tex.doRisky(test);
      System.out.println("end try");
    }
    catch (ScaryException se) { System.out.println("scary exception"); }
    finally { System.out.println("finally"); }
    System.out.println("end of main");
  }
  public void doRisky(String test) throws ScaryException {
    System.out.println("start risky");
    if ("yes".equals(test)) { throw new ScaryException(); }
    System.out.println("end risky");
  }
}
```

What is the output if this line is **String test = "yes";** ?

… and things for you to try out!

Assertions in Java (1/2)

- Assertions:
  - Java statements that enable you to assert (or verify) an assumption about your program.
  - Contain a Boolean expression that should be true during program execution.
- Assertions are used to ensure program correctness and avoid logic errors.
  - For internal consistency checks, e.g. to check the object state following mutation (due to a *setter* method being called).
  - During development (to *enable debugging*) but usually removed in production versions, e.g. via a run-time option.

Instead of using **`System.out.println()`**: assertions are more efficient and less error-prone.

Assertions in Java (2/2)

- Java assertions are declared via an **`assert`** statement, in either of two forms:

  (1) **`assert`** *`assertion-expression`* **OR**

  (2) **`assert`** *`assertion-expression : detailMessage`*

  – The ***`assertion-expression`*** expresses something that should be *true* at this point.

  – The ***`detailMessage`*** is a primitive type or an **`Object`** value.

  – An **`AssertionError`** exception is thrown if the assertion is *false*.

(1) no-argument constructor is used;
(2) appropriate constructor is used to match the message data type.

subclass of **`Error`**, so when an assertion is false, the program displays a message on the console and exits

Examples: Using `assert` Statement

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

No errors will be thrown, as both assertions are true (**i=10** and **sum=45**).

java.lang.AssertionError will be thrown with message **i is 11**, as the assertion is false.

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

Enabling and Disabling Assertions

- Assertions are disabled by default, at runtime. But you can always,
  - enable your program to run with assertions by calling it with the **`-enableassertions`** (***or*** in short form, **`-ea`**) switch;
  - disable your program from running with assertions by calling it with the **`-disableassertions`** (***or*** in short form, **`-da`**) switch;
  - enable/disable assertions at *package level* and at *class level*.
- Examples:

```
java -ea AssertionDemo
java -da Test
java -ea:ClassUsedByTest Test
java -da:ClassUsedByAssertionDemo AssertionDemo
```

From Java 5.0, running with assertions is disabled by default.

Guidelines for Assertions / Error Recovery and Avoidance

- Assertions:
  - Are not an alternative to throwing exceptions (which are to do with program *robustness*), they are to ensure the program's correctness.
  - Can be used for internal checks.
  - Are usually 'removed' from production code.
  - Should not be used to check the validity of a public method's argument(s).
  - Do not include normal functionality e.g.

```
// Incorrect use of assertions:
assert book.remove(name) != null;
```

Don't create assertions that change an object's state.

- Recovering from errors: client code should take note of error notifications. This means that it needs to,
  - Check return values.
  - Not 'ignore' exceptions.
  - Include code to attempt recovery: this will often require a loop.
- Avoiding errors: client code can often use server query methods to avoid errors. This means that,
  - Unchecked exceptions can be used.
  - Client logic is simplified.

Exercise 3

- What happens when you run the program as follows:
  - `java Foo`
  - `java –ea Foo`

```
public class Foo {
  public void m1(int value) {
    assert 0 <= value;
    System.out.println("OK");
  }
  public static void main(String[] args) {
    Foo foo = new Foo();
    System.out.print("foo.m1(1): ");
    foo.m1(1);
    System.out.print("foo.m1(-1): ");
    foo.m1(-1);
  }
}
```