Excerpt from the forthcoming Second Edition of *A Programmer's Guide to Java Certification: A Comprehensive Primer*

Khalid A. Mughal
Rolf W. Rasmussen

Publisher: Addison-Wesley
http://www.ii.uib.no/~khalid/pgjc/jcbook/
(Dated: 14-January-2003)

5.10 Assertions

Assertions in Java can be used to document and validate assumptions made about the state of the program at designated locations in the code. Each assertion contains a boolean expression which is expected to be `true` when the assertion is executed. If this assumption is `false`, the system throws a special assertion error. The assertion facility uses the exception handling mechanism to propagate the error. The assertion facility can be enabled or disabled at runtime.

The assertion facility is an invaluable aid in implementing *correct* programs, i.e. programs that adhere to their specification. It should not be confused with the exception handling mechanism which aids in developing *robust* programs, i.e. programs that handle unexpected conditions gracefully. Used judiciously, the two mechanisms facilitate programs that are *reliable*.

`assert` Statement and `AssertionError` Class

The following two forms of the `assert` statement can be used to specify assertions:

```
assert <boolean expression> ;                         // the simple form
assert <boolean expression> : <message expression> ;      // the augmented form
```

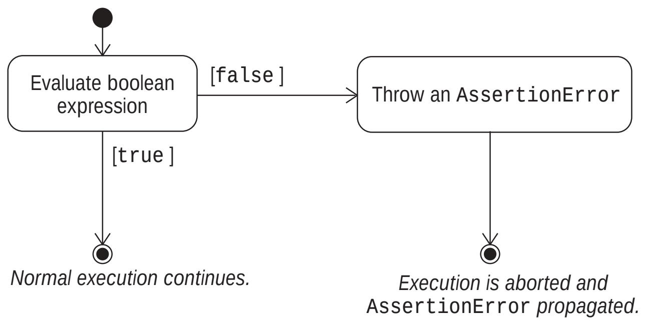
If assertions are enabled (p. 199), the execution of an assert statement proceeds as shown in Figure 5.8. The two forms are essentially equivalent to the following code respectively:

```
if (<assertions enabled> && !<boolean expression>)   // the simple form
    throw new AssertionError();
```

```
if (<assertions enabled> && !<boolean expression>)   // the augmented form
    throw new AssertionError(<message expression>);
```

If assertions are enabled, then *<boolean expression>* is evaluated. If its value is `true`, execution continues normally after the `assert` statement. However, if it is `false`, an `AssertionError` is thrown and propagated. In the simple form, the `AssertionError` does not provide any detailed message about the assertion failure.

Figure 5.8 *Execution of the simple* `assert` *statement (when assertions are enabled)*


Evaluate boolean expression
[false]
Throw an AssertionError
[true]
Normal execution continues.
Execution is aborted and AssertionError propagated.

The augmented form provides a *<message expression>* which can be used to provide a detailed error message. In the augmented form, if the assertion is `false`, the *<message expression>* is evaluated and its value passed to the appropriate `AssertionError` constructor. The *<message expression>* must evaluate to a value, i.e. either a primitive or a reference value. The `AssertionError` constructor invoked converts the value to a textual representation. In particular, the *<message expression>* cannot call a method that is declared `void`. The compiler will flag this as an error.

Lines (2), (3) and (4) in class `Speed` (Example 5.15) are assertion statements. In this particular context of calculating the speed, it is required that the values fulfil the criteria in lines (2), (3) and (4) in the private method `calcSpeed()`. Lines (2) and (4) use the simple form:

```
assert distance >= 0.0;                                    // (2)
...
assert speed >= 0.0;                                       // (4)
```

Line (3) uses the augmented form:

```
assert time > 0.0 : "Time is not a positive value: " + time;  // (3)
```

Line (3) is equivalent to the following line of code, assuming assertions have been enabled at runtime:

```
if (time <= 0.0) throw new AssertionError("Time is not a positive value: " + time);
```

The `java.lang.AssertionError` class is a subclass of `java.lang.Error` (Figure 5.6). This makes `AssertionError` exceptions unchecked. They could be explicitly caught and handled using the `try-catch` construct. The execution would then continue normally, as one would expect. However, since `Error` exceptions are seldom caught and handled by the program, the same applies to `AssertionError` exceptions. Catching these exceptions would defeat the whole purpose of the assertion facility.

In addition to the default constructor (which is invoked by the simple `assert` form), the `AssertionError` class provides seven single-parameter constructors: six for the primitive datatypes and one for object references. The type of the <*message expression*> used in the augmented assertion statement determines which of the overloaded constructors is invoked. It is not possible to query the `AssertionError` object for the actual value passed to the constructor. However, the method `getMessage()` will return the textual representation of the value.

Example 5.15 *Assertions*

```
public class Speed {

    public static void main(String[] args) {
        Speed objRef = new Speed();
        double speed = objRef.calcSpeed(-12.0, 3.0);                // (1a)
        // double speed = objRef.calcSpeed(12.0, -3.0);             // (1b)
        // double speed = objRef.calcSpeed(12.0, 2.0);              // (1c)
        // double speed = objRef.calcSpeed(12.0, 0.0);              // (1d)
        System.out.println("Speed (km/h): " + speed);
    }

    /** Requires distance >= 0.0 and time > 0.0 */
    private double calcSpeed(double distance, double time) {
        assert distance >= 0.0;                                      // (2)
        assert time >0.0 : "Time is not a positive value: " + time;  // (3)
        double speed = distance / time;
        assert speed >= 0.0;                                         // (4)
        return speed;
    }
}
```

Compiling Assertions

The assertion facility was introduced in J2SE 1.4. At the same time, two new options for the javac compiler were introduced for dealing with assertions in the source code:

- Option -source 1.4
  The javac compiler distributed with the Java SDK v1.4 will only compile assertions if the option -source 1.4 is used on the command-line:

  ```
  >javac -source 1.4 Speed.java
  ```

  This also means that incorrect use of the keyword assert will be flagged as an *error*, for example if assert is used as an identifier. The following program

  ```
  public class Legacy {
      public static void main(String[] args) {
          int assert = 2003;
          System.out.println("The year is: " + assert);
      }
  }
  ```

  when compiled, results in two errors:

  ```
  >javac -source 1.4 Legacy.java
  Legacy.java:4: as of release 1.4, assert is a keyword, and may not be used as an
   identifier
          int assert = 2003;
              ^
  Legacy.java:5: as of release 1.4, assert is a keyword, and may not be used as an
   identifier
          System.out.println("The year is: " + assert);
                                               ^
  2 errors
  ```

- Option -source 1.3
  The default behavior of the javac compiler is equivalent to using the option -source 1.3 on the command-line.

  ```
  >javac -speed 1.3 Speed.java
  Speed.java:14: warning: as of release 1.4, assert is a keyword, and may not be used
  as an identifier
          assert distance >= 0.0;                                 // (2)
          ^
  Speed.java:15: ';' expected
          assert distance >= 0.0;                                 // (2)
                          ^
  ...
  9 errors
  3 warnings
  ```

  The compiler will reject assert statements. It will also warn about the use of the keyword assert as an identifier. In other words, source code that contains the keyword assert as an identifier will compile (barring any other errors), but it will also result in a *warning*. Compiling and running the Legacy class above gives the following results:

```
>javac -source 1.3 Legacy.java
Legacy.java:4: as of release 1.4, assert is a keyword, and may not be used as an
 identifier
        int assert = 2003;
            ^
Legacy.java:5: as of release 1.4, assert is a keyword, and may not be used as an
 identifier
        System.out.println("The year is: " + assert);
                                             ^
2 warnings
>java Legacy
The year is: 2003
```

Runtime Enabling and Disabling of Assertions

Enabling assertions means they will be executed at runtime. By default, assertions are disabled. Their execution is then effectively equivalent to empty statements. This means that disabled assertions carry insignificant performance penalty, although they add storage overhead to the byte code of a class. Typically, assertions are enabled during development and left disabled once the program is deployed.

Two switches are provided to enable and disable assertions with various granularities in a `java` command. The switch `-enableassertions`, or its short form `-ea`, enables assertions, and the switch `-disableassertions`, or its short form `-da`, disables assertions at various granularities. The granularities that can be specified are shown in Table 5.2. *Assertion execution for all non-system classes*

Table 5.2 *Granularities for enabling and disabling assertions at runtime*

Argument	**Granularity**
`-ea` `-da`	Applies to all non-system classes.

`-ea:`*`<package name>`*`...`
`-da:`*`<package name>`*`...`
	Applies to the named package and its *subpackages*.

`-ea:...` `-da:...`	Applies to the unnamed package in the current working directory.

`-ea:`*`<class name>`*
`-da:`*`<class name>`*
	Applies to the named class.

The `-ea` option means that *all non-system* classes loaded during the execution of the program have their assertions enabled. A *system class* is a class that is in the Java platform libraries. For example, classes in the `java.*` packages are system classes. A system class is loaded directly by the JVM.

Note that class files not compiled with a J2SE 1.4-compatible compiler are not affected, whether assertions are enabled or disabled. Also, once a class has been loaded and initialized at runtime, its assertion status cannot be changed.

Assuming that the file `Speed.java` has been compiled with the `-source 1.4` option, all assertions in non-system classes required for execution (of which `Speed` class is one) can be enabled, and the program run as follows:

```
>java -ea Speed
java.lang.AssertionError
      at Speed.calcSpeed(Speed.java:14)
      at Speed.main(Speed.java:6)
Exception in thread "main"
```

Since the distance is negative in line (1a), the assertion in line (2) fails in Example 5.15. An `AssertionError` is thrown, which is propagated, being finally caught by the default exception handler and resulting in the stack trace being printed on the terminal.

All assertions (in all non-system classes) can be disabled during the execution of the `Speed` class:

```
>java -da Speed
Speed (km/h): -4.0
```

In this case, this is effectively equivalent to running the program with neither the `-ea` nor the `-da` options:

```
>java Speed
Speed (km/h): -4.0
```

If we comment out line (1a) and uncomment line (1b) in Example 5.15, and run the program with the options enabled, we get the following behavior from the program:

```
>java -ea Speed
java.lang.AssertionError: Time is not a positive value: -3.0
      at Speed.calcSpeed(Speed.java:15)
      at Speed.main(Speed.java:7)
Exception in thread "main"
```

We see that the value of the <*message expression*> in the augmented assertion in line (3) is written on the terminal, together with the stack trace, because this assertion failed. The augmented form is recommended, as it allows a detailed error message to be included in reporting the assertion failure.

Assertion execution at the package level

Assume that we have a program called `Trickster` in the unnamed package, that uses the package hierarchy for the `wizard` package shown in Figure 4.3 on p. 137.

The following command-line will only enable assertions for all classes in the package `wizard.pandorasBox` and its subpackage `wizard.pandorasBox.artifacts`. The assertions in the class `Trickster` are not enabled :

```
>java -ea:wizard.pandorasBox... Trickster
```

Without the ... notation, the package name will be interpreted as a class name (see below). Non-existent package names specified in the command-line are silently accepted, but simply have no consequences under execution.

The following command-line will only enable assertions in the unnamed package, and thereby the assertions in the class `Trickster`, since this class resides in the unnamed package:

```
>java -ea:... Trickster
```

Note that the package switch applies to the package specified and all its subpackages, recursively.

Assertion execution at the class level

The following command-line will only enable assertions in the `Trickster` class:

```
>java -ea:Trickster Tricksters
```

The following command-line will only enable assertions in the named class `wizard.pandorasBox.artifacts.Ailment` and no other class:

```
>java -ea:wizard.pandorasBox.artifacts.Ailment Trickster
```

A `java` command can contain multiple instances of the switches, each specifying its own granularity. The switches are then processed in order of their specification from left to right, before any classes are loaded. The latter switches take priority over former switches. This allows a fine-grained control of what assertions are enabled at runtime. The following command-line will enable assertions for all classes in the package `wizard.pandorasBox` and its subpackage `wizard.pandorasBox.artifacts`, but disable them in the class `wizard.pandorasBox.artifacts.Ailment`:

```
>java -ea:wizard.pandorasBox... -da:wizard.pandorasBox.artifacts.Ailment Trickster
```

The following switches all enable assertions in the class `wizard.spells.Baldness`:

```
>java -ea                          Trickster
>java -ea:wizard...                Trickster
>java -ea:wizard.spells...         Trickster
>java -ea:wizard.spells.Baldness   Trickster
```

It is worth noting that inheritance (Section 6.1, p. 240) has no affect on the execution of assertions. Assertions are enabled or disabled on per-class basis. Whether assertions in the superclass will be executed through code inherited by the subclass, depends entirely on the superclass. In the following command-line, assertions from the superclass `wizard.pandorasBox.artifacts.Ailment` will not be executed, although assertions for the subclass `wizard.spells.Baldness` are enabled:

```
>java -ea -da:wizard.pandorasBox.artifacts.Ailment Trickster
```

Assertion execution for all system classes

In order to enable or disable assertions in *all system classes*, we can use the switches shown in Table 5.3. Enabling assertions in system classes can be useful to shed light on internal errors reported by the JVM. In the following command-line, the first switch will enable assertions for all system classes. The second switch will enable assertions in the package `wizard` and its subpackages `wizard.pandorasBox`, `wizard.pandorasBox.artifacts` and `wizard.spells`, but the third switch will disable them in the package `wizard.pandorasBox.artifacts`:

```
>java -esa -ea:wizard... -da:wizard.pandorasBox.artifacts... Trickster
```

Table 5.3 *Options for enabling and disabling assertions in all system classes at runtime*

Option	Short form	Description
`-enablesystemassertions`	`-esa`	Enable assertions in *all* system classes.
`-disablesystemassertions`	`-dsa`	Disable assertions in *all* system classes.

Using Assertions

Assertions should have no side effects which can produce adverse behavior in the code once they are disabled. The assertion facility is a defensive mechanism, meaning that it should only be used to test the code, and should not be employed after the code is delivered. The program should exhibit the same behavior whether assertions are enabled or disabled. The program should not rely on any computations done within an assertion statement. With assertions enabled, the following statement would be executed:

```
assert reactor.controlCoreTemperature();
```

but if assertions were disabled, it could have dire consequences.

Assertions should also not be used to validate information supplied by a client. A typical example is argument checking in public methods. Argument checking is part of such a method’s contract, which could be violated if the assertions were disabled. Another drawback is that assertion failures can only provide limited information, in the form of an `AssertionError`, about the cause of any failure. Appropriate argument checking can provide more suitable information about erroneous arguments, in the form of specific exceptions such as `IllegalArgumentException`, `IndexOutOfBoundsException` or `NullPointerException`.

The rest of this section illustrates useful idioms that employ assertions.

Internal Invariants

Very often assumptions about the program are documented as comments in the code. The following code makes the assumption, in line (1), that variable `status` must be negative for the `else` clause to be executed:

```
int status = ref1.compareTo(ref2);
if (status == 0) {
    ...
} else if (status > 0) {
    ...
} else { // (1) status must be negative.
    ...
}
```

This assumption is an *internal invariant* and can be verified using an assertion, as shown in line (2) below:

```
int status = ref1.compareTo(ref2);
if (status == 0) {
    ...
} else if (status > 0) {
    ...
} else {
    assert status < 0 : status; // (2)
    ...
}
```

Often an alternative action is chosen, based on a value which is guaranteed to be one of a small set of predefined values. A `switch` statement with no `default` clause is a typical example. The value of the `switch` expression is guaranteed to be one of the `case` labels, and the `default` case is omitted, as the following code shows:

```
switch (trinityMember) {
    case Housefather:
        ...
        break;
    case THE_SON:
        ...
        break;
    case THE_HOLY_GHOST:
        ...
        break;
}
```

A `default` clause that executes an assertion can be used to formulate this invariant:

```
     default:
         assert false : trinityMember;
```

If assertions are enabled, an `AssertionError` will signal the failure in case the trinity no longer holds.

However, the above idiom causes a compile time error in a non-`void` method if all `case` labels return a value and no `return` statement follows the `switch` statement:

```
    switch (trinityMember) {
        case THE_FATHER:
            return psalm101;
        case THE_SON:
            return psalm102;
        case THE_HOLY_GHOST:
            return psalm103;
        default:
            assert false: trinityMember;
    }
    return psalm100;        // (3) Compile time error if commented out.
```

Without line (3) and with assertions disabled, the method could return without a value, violating the fact that it is a non-`void` method. Explicitly throwing an `AssertionError` rather than using an `assert` statement in the `default` clause, would be a better option in this case:

```
        default:
            throw new AssertionError(trinityMember);
```

Control Flow Invariants

Control flow invariants can be used to test assumptions about the flow of control in the program. The following idiom can be employed to explicitly test that certain locations in the code will never be reached:

```
        assert false : "This line should never be reached.";
```

If program control does reach this statement, assertion failure will detect it.

In the following code, the assumption is that execution never reaches the end of the method declaration indicated by line (1):

```
    private void securityMonitor() {
        // ...
        while (alwaysOnDuty) {
            // ...
            if (needMaintenance)
                return;
            // ...
        }
        // (1) This line should never be reached.
    }
```

The assertion above can be inserted after the comment at line (1) to check the assumption.

Care should be taken in using this idiom, as the compiler can flag the `assert` statement at this location as being unreachable. For example, if the compiler can deduce that the `while` condition will always be `true`, it will flag the `assert` statement as being unreachable.

Preconditions and Postconditions

The assertion facility can be used to practise a limited form of *programming-by-contract*. For example, the assertion facility can be used to check that methods comply with their contract.

Preconditions define assumptions for the proper execution of a method when it is invoked. As discussed earlier, preconditions on public methods should not be checked using assertions. For non-public methods, preconditions can be checked at the start of method execution:

```
private void adjustReactorThroughput(int increment) {
    // Precondition:
    assert isValid(increment) : "Throughput increment invalid.";
    // Proceed with the adjustment.
    // ...
}
```

Section 9.4 (p. 349) provides an example of a *lock-status* precondition in a non-synchronized method, where an assertion is used to check whether the current thread holds a lock on a certain object.

Postconditions define assumptions about the successful completion of a method. Postconditions in any method can be checked by assertions executed just before returning from the method. For example, if the method `adjustReactorThroughPut()` guarantees that the reactor core is in a stable state after its completion, we can check this postcondition using an assertion:

```
private void adjustReactorThroughput(int increment) {
    // Precondition:
    assert isValid(increment) : "Throughput increment invalid.";
    // Proceed with the adjustment.
    // ...
    // Postcondition -- the last action performed before returning.
    assert isCoreStable() : “Reactor core not stable.”;
}
```

Section 7.4 (p. 306) provides an example using a *local class* where data can be saved before doing a computation, so that it can later be used to check a postcondition.

Other uses

If minimizing the size of the class file is crucial, then the conditional compilation idiom should be used to insert assertions in the source code:

```
static final boolean compileAsserts = false;
...
if (compileAsserts)
   assert whatEverYouWant;         // Not compiled if compileAsserts is false.
...
```

It is possible to enforce that a class be loaded and initialized only if its assertions are enabled. The idiom for this purpose uses a *static initializer* (Section 8.2, p. 326):

```
static {  // Static initializer
    boolean assertsAreEnabled = false;  // (1)
    assert assertsAreEnabled = true;    // (2) utilizing side effect
    if (!assertsAreEnabled)             // (3)
        throw new AssertionError("Enable assertions!");
}
```

Line (1) sets the local variable `assertsAreEnabled` to `false`. If assertions are enabled, line (2) is executed. The assignment operator sets the variable `assertsAreEnabled` to `true` as a side effect of evaluating the boolean expression which has the value `true`. The assertion in line (2) is of course `true`. No exception is thrown by the `if` statement in line (3). However, if assertions are disabled, line (2) is never executed. As the variable `assertsAreEnabled` is `false`, the `if` statement in line (3) throws an exception. The static initializer is placed first in the class declaration, so that it is executed first during class initialization.

Review questions

5.24 Assuming assertions are enabled, which of these assertion statements will throw an error?

Select the two correct answers.

(a) `assert true : true;`
(b) `assert true : false;`
(c) `assert false : true;`
(d) `assert false : false;`

5.25 Which of the following are valid runtime options?

Select the two correct answers.

(a) `-ae`
(b) `-enableassertions`
(c) `-source 1.4`
(d) `-disablesystemassertions`
(e) `-dea`

5.26 What is the class name of the exception thrown by an assertion statement?

Select the one correct answer.

(a) Depends on the assertion statement.
(b) `FailedAssertion`
(c) `AssertionException`
(d) `RuntimeException`
(e) `AssertionError`
(f) `Error`

5.27 What can cause an assertion statement to be ignored?

Select the one correct answer.

(a) Nothing.
(b) Using appropriate compiler options.
(c) Using appropriate runtime options.
(d) Using both appropriate compiler and runtime options.

5.28 Given the following method, which of these statements will throw an exception, assuming assertions are enabled?

```
static int inv(int value) {
    assert value > -50 : value < 100;
    return 100/value;
}
```

Select the two correct answers.

(a) `inv(-50);`
(b) `inv(0);`
(c) `inv(50);`
(d) `inv(100);`
(e) `inv(150);`

5.29 Which runtime options would cause assertions to be enabled for the class `org.example.ttp.Bottle`?

Select the two correct answers.

(a) `-ea`
(b) `-ea:Bottle`
(c) `-ea:org.example`
(d) `-ea:org...`
(e) `-enableexceptions:org.example.ttp.Bottle`
(f) `-ea:org.example.ttp`

5.30 What will be the result of compiling and running the following code with assertions enabled?

```
public class TernaryAssertion {
    public static void assertBounds(int low, int high, int value) {
        assert ( value > low ? value < high : false )
            : (value < high ? "too low" : "too high" );
    }
    public static void main(String[] args) {
        assertBounds(100, 200, 150);
    }
}
```

Select the one correct answer.

(a) The compilation fails, because the method name `assertBounds` cannot begin with the keyword `assert`.
(b) The compilation fails because the `assert` statement is invalid.
(c) The compilation succeeds and the program runs without errors.
(d) The compilation succeeds and an `AssertionError` with the error message `"too low"` is thrown.
(e) The compilation succeeds and an `AssertionError` with the error message `"too high"` is thrown.

5.31 Which of the following statements about the `AssertionError` class are true?

Select the two correct answers.

(a) It is a checked exception.
(b) It has a method named `toString()`.
(c) It has a method named `getErrorMessage()`.
(d) It can be caught by a `try-catch` construct.

5.32 Which of these classes is the direct superclass of `AssertionError`?

Select the one correct answer.

(a) `Object`
(b) `Throwable`
(c) `Exception`
(d) `Error`
(e) `RuntimeError`

5.33 Given the following command, which classes would have assertions enabled?

```
java -ea -da:com... net.example.LaunchTranslator
```

Select the two correct answers.

(a) `com.example.Translator`
(b) `java.lang.String`
(c) `dot.com.Boom`
(d) `net.example.LaunchTranslator`
(e) `java.lang.AssertionError`

Annotated Answers to Review Questions

5.24 *(c) and (d)*

Statements (c) and (d) will throw an error because the first expression is `false`. Statement (c) will report `true` as the error message, while (d) will report `false` as the error message.

5.25 *(b) and (d)*

5.26 *(e)*

The class of exceptions thrown by assertion statements is always `AssertionError`.

5.27 *(c)*

Assertions can be enabled or disabled at runtime, but the `assert` statements are always compiled into bytecode.

5.28 *(a) and (b)*

Statement (a) will cause the `assert` statement to throw an `AssertionError`, since `(-50 > -50)` evaluates to `false`. Statement (b) will cause the expression `100/value` to throw an `ArithmeticException`, since an integer division by zero is attempted.

5.29 *(a) and (d)*

Option (a) enables assertions for all non-system classes, while (d) enables assertions for all classes in package `org` and its subpackages. Options (b), (c) and (f) try to enable assertions in specifically named classes: `Bottle`, `org.example` and `org.example.ttp`. Option (e) is not a valid runtime option.

5.30 *(c)*

The `assert` statement rightly asserts that 150 is greater than 100 and less than 200.

5.31 *(b) and (d)*

The `AssertionError` class, like all other error classes, is not a checked exception, and not need to be declared using a `throws`-clause. After an `AssertionError` is thrown, it is propagated exactly the same way as other exceptions, and can be caught by a `try`-`catch` construct.

5.32 *(d)*

5.33 *(c) and (d)*

The command line enables assertions for all non-system classes, except for those in the `com` package or one of its subpackages. Assertions are not enabled for the system classes in (b) and (e).