Java SE Documentation

# Programming With Assertions

An *assertion* is a statement in the Java programming language that enables you to test your assumptions about your program. For example, if you write a method that calculates the speed of a particle, you might assert that the calculated speed is less than the speed of light.

Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.

Experience has shown that writing assertions while programming is one of the quickest and most effective ways to detect and correct bugs. As an added benefit, assertions serve to document the inner workings of your program, enhancing maintainability.

This document shows you how to program with assertions. It covers the topics:



## Introduction

The assertion statement has two forms. The first, simpler form is:

```
assert Expression1 ;
```

where *Expression*$_1$ is a `boolean` expression. When the system runs the assertion, it evaluates *Expression*$_1$ and if it is `false` throws an `AssertionError` with no detail message.

The second form of the assertion statement is:

`assert` *Expression*$_1$ `:` *Expression*$_2$ `;`

where:

- *Expression*$_1$ is a boolean expression.
- *Expression*$_2$ is an expression that has a value. (It cannot be an invocation of a method that is declared `void`.)

Use this version of the `assert` statement to provide a detail message for the `AssertionError`. The system passes the value of *Expression*$_2$ to the appropriate `AssertionError` constructor, which uses the string representation of the value as the error's detail message.

The purpose of the detail message is to capture and communicate the details of the assertion failure. The message should allow you to diagnose and ultimately fix the error that led the assertion to fail. Note that the detail message is *not* a user-level error message, so it is generally unnecessary to make these messages understandable in isolation, or to internationalize them. The detail message is meant to be interpreted in the context of a full stack trace, in conjunction with the source code containing the failed assertion.

Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown. The second form of the assertion statement should be used in preference to the first only when the program has some additional information that might help diagnose the failure. For example, if *Expression*$_1$ involves the relationship between two variables `x` and `y`, the second form should be used. Under these circumstances, a reasonable candidate for *Expression*$_2$ would be `"x: " + x + ", y: " + y`.

In some cases *Expression*$_1$ may be expensive to evaluate. For example, suppose you write a method to find the minimum element in an unsorted list, and you add an assertion to verify that the selected element is indeed the minimum. The work done by the assert will be at least as expensive as the work done by the method itself. To ensure that assertions are not a performance liability in deployed applications, assertions can be enabled or disabled when the program is started, and are disabled by default. Disabling assertions eliminates their performance penalty entirely. Once disabled, they are essentially equivalent to *empty statements* in semantics and performance. See Enabling and Disabling Assertions for more information.

The addition of the `assert` keyword to the Java programming language has implications for existing code. See Compatibility With Existing Programs for more information.

## Putting Assertions Into Your Code

There are many situations where it is good to use assertions, including:

- Internal Invariants
- Control-Flow Invariants
- Preconditions, Postconditions, and Class Invariants

There are also situations where you should *not* use them:

- Do *not* use assertions for argument checking in public methods.
  Argument checking is typically part of the published specifications (or *contract*) of a method, and these specifications must be obeyed whether assertions are enabled or disabled. Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException`, or `NullPointerException`). An assertion failure will not throw an appropriate exception.

- Do *not* use assertions to do any work that your application requires for correct operation.

Because assertions may be disabled, programs must not assume that the boolean expression contained in an assertion will be evaluated. Violating this rule has dire consequences. For example, suppose you wanted to remove all of the null elements from a list `names`, and knew that the list contained one or more nulls. It would be wrong to do this:

```
// Broken! - action is contained in assertion
assert names.remove(null);
```

The program would work fine when asserts were enabled, but would fail when they were disabled, as it would no longer remove the null elements from the list. The correct idiom is to perform the action before the assertion and then assert that the action succeeded:

```
// Fixed - action precedes assertion
boolean nullsRemoved = names.remove(null);
assert nullsRemoved;  // Runs whether or not asserts are enabled
```

As a rule, the expressions contained in assertions should be free of *side effects*: evaluating the expression should not affect any state that is visible after the evaluation is complete. One exception to this rule is that assertions can modify state that is used only from within other assertions. An idiom that makes use of this exception is presented later in this document.

## Internal Invariants

Before assertions were available, many programmers used comments to indicate their assumptions concerning a program's behavior. For example, you might have written something like this to explain your assumption about an `else` clause in a multiway if-statement:

```
if (i % 3 == 0) {
    ...
} else if (i % 3 == 1) {
    ...
} else { // We know (i % 3 == 2)
    ...
}
```

You should now **use an assertion whenever you would have written a comment that asserts an invariant**. For example, you should rewrite the previous if-statement like this:

```
if (i % 3 == 0) {
   ...
} else if (i % 3 == 1) {
    ...
} else {
    assert i % 3 == 2 : i;
    ...
}
```

Note, incidentally, that the assertion in the above example may fail if `i` is negative, as the `%` operator is not a true *modulus* operator, but computes the *remainder*, which may be negative.

Another good candidate for an assertion is a `switch` statement with no `default` case. The absence of a `default` case typically indicates that a programmer believes that one of the cases will always be executed. The assumption that a particular variable will have one of a small number of values is an invariant that should be checked with an assertion. For example, suppose the following `switch` statement appears in a program that handles playing cards:

```
switch(suit) {
  case Suit.CLUBS:
    ...
  break;

  case Suit.DIAMONDS:
    ...
  break;

  case Suit.HEARTS:
    ...
    break;

  case Suit.SPADES:
      ...
}
```

It probably indicates an assumption that the `suit` variable will have one of only four values. To test this assumption, you should add the following default case:

```
default:
    assert false : suit;
```

If the `suit` variable takes on another value and assertions are enabled, the assert will fail and an `AssertionError` will be thrown.

An acceptable alternative is:

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

This alternative offers protection even if assertions are disabled, but the extra protection adds no cost: the `throw` statement won't execute unless the program has failed. Moreover, the alternative is legal under some circumstances where the `assert` statement is not. If the enclosing method returns a value, each case in the `switch` statement contains a `return` statement, and no `return` statement follows the `switch` statement, then it would cause a syntax error to add a default case with an assertion. (The method would return without a value if no case matched and assertions were disabled.)

## Control-Flow Invariants

The previous example not only tests an invariant, it also checks an assumption about the application's flow of control. The author of the original `switch` statement probably assumed not only that the `suit` variable would always have one of four values, but also that one of the four cases would always be executed. It points out another general area where you should use assertions: **place an assertion at any location you assume will not be reached**. The assertions statement to use is:

```
assert false;
```

For example, suppose you have a method that looks like this:

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    // Execution should never reach this point!!!
}
```

Replace the final comment so that the code now reads:

```
void foo() {
    for (...) {
      if (...)
        return;
    }
    assert false; // Execution should never reach this point!
}
```

**Note:** Use this technique with discretion. If a statement is unreachable as defined in the Java Language Specification, you will get a compile time error if you try to assert that it is not reached. Again, an acceptable alternative is simply to throw an `AssertionError`.

## Preconditions, Postconditions, and Class Invariants

While the assert construct is not a full-blown *design-by-contract* facility, it can help support an informal design-by-contract style of programming. This section shows you how to use asserts for:

- Preconditions — what must be true when a method is invoked.
  - Lock-Status Preconditions — preconditions concerning whether or not a given lock is held.
- Postconditions — what must be true after a method completes successfully.
- Class invariants — what must be true about each instance of a class.

### Preconditions

By convention, preconditions on *public* methods are enforced by explicit checks that throw particular, specified exceptions. For example:

```
/**
  * Sets the refresh rate.
  *
  * @param  rate refresh rate, in frames per second.
  * @throws IllegalArgumentException if rate <= 0 or
  * rate > MAX_REFRESH_RATE.
*/
public void setRefreshRate(int rate) {
  // Enforce specified precondition in public method
```

```
  if (rate <= 0 || rate > MAX_REFRESH_RATE)
    throw new IllegalArgumentException("Illegal rate: " + rate);
    setRefreshInterval(1000/rate);
  }
```

This convention is unaffected by the addition of the `assert` construct. **Do not use assertions to check the parameters of a public method.** An assert is inappropriate because the method guarantees that it will always enforce the argument checks. It must check its arguments whether or not assertions are enabled. Further, the `assert` construct does not throw an exception of the specified type. It can throw only an `AssertionError`.

You can, however, use an assertion to test a *nonpublic* method's precondition that you believe will be true no matter what a client does with the class. For example, an assertion *is* appropriate in the following "helper method" that is invoked by the previous method:

```
/**
 * Sets the refresh interval (which must correspond to a legal frame rate).
 *
 * @param  interval refresh interval in milliseconds.
*/
 private void setRefreshInterval(int interval) {
  // Confirm adherence to precondition in nonpublic method
  assert interval > 0 && interval <= 1000/MAX_REFRESH_RATE : interval;

  ... // Set the refresh interval
 }
```

Note, the above assertion will fail if `MAX_REFRESH_RATE` is greater than 1000 and the client selects a refresh rate greater than 1000. This would, in fact, indicate a bug in the library!

### Lock-Status Preconditions

Classes designed for multithreaded use often have non-public methods with preconditions relating to whether or not some lock is held. For example, it is not uncommon to see something like this:

```
private Object[] a;
public synchronized int find(Object key) {
  return find(key, a, 0, a.length);
}

// Recursive helper method - always called with a lock on this object
private int find(Object key, Object[] arr, int start, int len) {
 ...
}
```

A static method called `holdsLock` has been added to the `Thread` class to test whether the current thread holds the lock on a specified object. This method can be used in combination with an `assert` statement to supplement a comment describing a lock-status precondition, as shown in the following example:

```
// Recursive helper method - always called with a lock on this.
private int find(Object key, Object[] arr, int start, int len) {
  assert Thread.holdsLock(this); // lock-status assertion
  ...
}
```

Note that it is also possible to write a lock-status assertion asserting that a given lock *isn't* held.

### Postconditions

You can test postcondition with assertions in both public and nonpublic methods. For example, the following public method uses an `assert` statement to check a post condition:

```
 /**
  * Returns a BigInteger whose value is (this-1 mod m).
  *
  * @param  m the modulus.
  * @return this-1 mod m.
  * @throws ArithmeticException  m <= 0, or this BigInteger
  *has no multiplicative inverse mod m (that is, this BigInteger
  *is not relatively prime to m).
  */
public BigInteger modInverse(BigInteger m) {
  if (m.signum <= 0)
    throw new ArithmeticException("Modulus not positive: " + m);
  ... // Do the computation
  assert this.multiply(result).mod(m).equals(ONE) : this;
  return result;
}
```

Occasionally it is necessary to save some data prior to performing a computation in order to check a postcondition. You can do this with two `assert` statements and a simple inner class that saves the state of one or more variables so they can be checked (or rechecked) after the computation. For example, suppose you have a piece of code that looks like this:

```
 void foo(int[] array) {
  // Manipulate array
  ...

  // At this point, array will contain exactly the ints that it did
  // prior to manipulation, in the same order.
 }
```

Here is how you could modify the above method to turn the textual assertion of a postcondition into a functional one:

```
 void foo(final int[] array) {
```

```
  // Inner class that saves state and performs final consistency check
  class DataCopy {
private int[] arrayCopy;

DataCopy() { arrayCopy = (int[]) array.clone(); }

boolean isConsistent() { return Arrays.equals(array, arrayCopy); }
  }

  DataCopy copy = null;

  // Always succeeds; has side effect of saving a copy of array
  assert ((copy = new DataCopy()) != null);

  ... // Manipulate array

  // Ensure array has same ints in same order as before manipulation.
  assert copy.isConsistent();
  }
```

You can easily generalize this idiom to save more than one data field, and to test arbitrarily complex assertions concerning pre-computation and post-computation values.

You might be tempted to replace the first assert statement (which is executed solely for its side-effect) by the following, more expressive statement:

```
copy = new DataCopy();
```

Don't make this replacement. The statement above would copy the array whether or not asserts were enabled, violating the principle that assertions should have no cost when disabled.

### Class Invariants

A class invariant is a type of internal invariant that applies to every instance of a class at all times, except when an instance is in transition from one consistent state to another. A class invariant can specify the relationships among multiple attributes, and should be true before and after any method completes. For example, suppose you implement a balanced tree data structure of some sort. A class invariant might be that the tree is balanced and properly ordered.

The assertion mechanism does not enforce any particular style for checking invariants. It is sometimes convenient, though, to combine the expressions that check required constraints into a single internal method that can be called by assertions. Continuing the balanced tree example, it might be appropriate to implement a private method that checked that the tree was indeed balanced as per the dictates of the data structure:

```
// Returns true if this tree is properly balanced
private boolean balanced() {
 ...
}
```

Because this method checks a constraint that should be true before and after any method completes, each public method and constructor should contain the following line

immediately prior to its return:

```
assert balanced();
```

It is generally unnecessary to place similar checks at the head of each public method unless the data structure is implemented by native methods. In this case, it is possible that a memory corruption bug could corrupt a "native peer" data structure in between method invocations. A failure of the assertion at the head of such a method would indicate that such memory corruption had occurred. Similarly, it may be advisable to include class invariant checks at the heads of methods in classes whose state is modifiable by other classes. (Better yet, design classes so that their state is not directly visible to other classes!)

## Advanced Uses

The following sections discuss topics that apply only to resource-constrained devices and to systems where asserts must not be disabled in the field. If you have no interest in these topics, skip to the next section, "Compiling Files that Use Assertions".

### Removing all Trace of Assertions from Class Files

Programmers developing applications for resource-constrained devices may wish to strip assertions out of class files entirely. While this makes it impossible to enable assertions in the field, it also reduces class file size, possibly leading to improved class loading performance. In the absence of a high quality JIT, it could lead to decreased footprint and improved runtime performance.

The assertion facility offers no direct support for stripping assertions out of class files. The assert statement may, however, be used in conjunction with the "conditional compilation" idiom described in the Java Language Specification, enabling the compiler to eliminate all traces of these asserts from the class files that it generates:

```
static final boolean asserts = ... ; // false to eliminate asserts

if (asserts)
 assert <expr> ;
```

### Requiring that Assertions are Enabled

Programmers of certain critical systems might wish to ensure that assertions are not disabled in the field. The following static initialization idiom prevents a class from being initialized if its assertions have been disabled:

```
static {
 boolean assertsEnabled = false;
 assert assertsEnabled = true; // Intentional side effect!!!
 if (!assertsEnabled)
throw new RuntimeException("Asserts must be enabled!!!");
 }
```

Put this static-initializer at the top of your class.

## Compiling Files That Use Assertions

In order for the `javac` compiler to accept code containing assertions, you must use the `-source 1.4` command-line option as in this example:

```
javac -source 1.4 MyClass.java
```

This flag is necessary so as not to cause source compatibility problems.

## Enabling and Disabling Assertions

By default, assertions are disabled at runtime. Two command-line switches allow you to selectively enable or disable assertions.

To enable assertions at various granularities, use the `-enableassertions`, or `-ea`, switch. To disable assertions at various granularities, use the `-disableassertions`, or `-da`, switch. You specify the granularity with the arguments that you provide to the switch:

- no arguments
  Enables or disables assertions in all classes except system classes.
- *packageName*`...`
  Enables or disables assertions in the named package and any subpackages.
- `...`
  Enables or disables assertions in the unnamed package in the current working directory.
- *className*
  Enables or disables assertions in the named class

For example, the following command runs a program, `BatTutor`, with assertions enabled in only package `com.wombat.fruitbat` and its subpackages:

```
java -ea:com.wombat.fruitbat... BatTutor
```

If a single command line contains multiple instances of these switches, they are processed in order before loading any classes. For example, the following command runs the `BatTutor` program with assertions enabled in package `com.wombat.fruitbat` but disabled in class `com.wombat.fruitbat.Brickbat`:

```
java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat BatTutor
```

The above switches apply to all class loaders. With one exception, they also apply to *system classes* (which do not have an explicit class loader). The exception concerns the switches with no arguments, which (as indicated above) do not apply to system classes.This behavior makes it easy to enable asserts in all classes except for system classes, which is commonly desirable.

To enable assertions in all system classes, use a different switch: `-enablesystemassertions`, or `-esa`. Similarly, to disable assertions in system classes, use `-disablesystemassertions`, or `-dsa`.

For example, the following command runs the `BatTutor` program with assertions enabled in system classes, as well as in the `com.wombat.fruitbat` package and its subpackages:

```
java -esa -ea:com.wombat.fruitbat...
```

The assertion status of a class (enabled or disabled) is set at the time it is initialized, and does not change. There is, however, one corner case that demands special treatment. It is possible, though generally not desirable, to execute methods or constructors prior to initialization. This can happen when a class hierarchy contains a circularity in its static initialization.

If an `assert` statement executes before its class is initialized, the execution must behave as if assertions were enabled in the class. This topic is discussed in detail in the assertions specification in the Java Language Specification.

## Compatibility With Existing Programs

The addition of the `assert` keyword to the Java programming language does not cause any problems with preexisting binaries (`.class` files). If you try to compile an application that uses `assert` as an identifier, however, you will receive a warning or error message. In order to ease the transition from a world where `assert` is a legal identifier to one where it isn't, the compiler supports two modes of operation in this release:

- **source mode 1.3** (default) — the compiler accepts programs that use `assert` as an identifier, but issues warnings. In this mode, programs are *not* permitted to use the `assert` statement.
- **source mode 1.4** — the compiler generates an error message if the program uses `assert` as an identifier. In this mode, programs *are* permitted to use the `assert` statement.

Unless you specifically request source mode 1.4 with the `-source 1.4` flag, the compiler operates in source mode 1.3. *If you forget to use this flag, programs that use the new `assert` statement will not compile*. Having the compiler use the old semantics as its default behavior (that is, allowing `assert` to be used as an identifier) was done for maximal source compatibility. Source mode 1.3 is likely to be phased out over time.

## Design FAQ

Here is a collection of frequently asked questions concerning the design of the assertion facility.



## General Questions

- **Why provide an assertion facility, given that one can program assertions atop the Java programming language with no special support?**

  Although ad hoc implementations are possible, they are of necessity either ugly (requiring an `if` statement for each assertion) or inefficient (evaluating the condition even if assertions are disabled). Further, each ad hoc implementation has its own means of enabling and disabling assertions, which lessens the utility of these implementations, especially for debugging in the field. As a result of these shortcomings, assertions have never become a part of the culture among engineers using the Java programming language. Adding assertion support to the platform stands a good chance of rectifying this situation.

- **Why does this facility justify a language change, as opposed to a library solution?**

  We recognize that a language change is a serious effort, not to be undertaken lightly. The library approach was considered. It was, however, deemed essential that the runtime cost of assertions be negligible if they are disabled. In order to achieve this with a library, the programmer is forced to hard-code each assertion as an `if` statement. Many programmers would not do this. Either they would omit the if statement and performance would suffer, or they would ignore the facility entirely. Note also that assertions were contained in James Gosling's original specification for the Java programming language. Assertions were removed from the Oak specification because time constraints prevented a satisfactory design and implementation.

- **Why not provide a full-fledged *design-by-contract* facility with preconditions, postconditions and class invariants, like the one in the Eiffel programming language?**

  We considered providing such a facility, but were unable to convince ourselves that it is possible to graft it onto the Java programming language without massive changes to the Java platform libraries, and massive inconsistencies between old and new libraries. Further, we were not convinced that such a facility would preserve the simplicity that is the hallmark of the Java programming language. On balance, we came to the conclusion that a simple boolean assertion facility was a fairly straight-forward solution and

far less risky. It's worth noting that adding a boolean assertion facility to the language doesn't preclude adding a full-fledged design-by-contract facility at some time in the future.

The simple assertion facility does enable a limited form of design-by-contract style programming. The `assert` statement is appropriate for nonpublic precondition, postcondition and class invariant checking. Public precondition checking should still be performed by checks inside methods that result in particular, documented exceptions, such as `IllegalArgumentException` and `IllegalStateException`.

- **In addition to boolean assertions, why not provide an assert-like construct to suppress the execution of an entire block of code if assertions are disabled?**
  Providing such a construct would encourage programmers to put complex assertions inline, when they are better relegated to separate methods.

## Compatibility

- **Won't the new keyword cause compatibility problems with existing programs that use `assert` as an identifier?**
  Yes, for source files. (Binaries for classes that use `assert` as an identifier will continue to work fine.) To ease the transition, we implemented a strategy whereby developers can continue using `assert` as an identifier during a transitional period.

- **Doesn't this facility produce class files that cannot be run against older JREs?**
  Yes. Class files will contain calls to the new `ClassLoader` and `Class` methods, such as `desiredAssertionStatus`. If a class file containing calls to these methods is run against an older JRE (whose `ClassLoader` class doesn't define the methods), the program will fail at run time, throwing a `NoSuchMethodError`. It is generally the case that programs using new facilities are not compatible with older releases.

## Syntax and Semantics

- **Why allow primitive types in *Expression*$_2$?**
  There is no compelling reason to restrict the type of this expression. Allowing arbitrary types provides convenience for developers who, for example, want to associate a unique integer code with each assertion. Further, it makes this expression feel like the argument of `System.out.println(...)`, which is seen as desirable.

## The AssertionError Class

- **When an `AssertionError` is generated by an assert statement in which *Expression*$_2$ is absent, why isn't the program text of the asserted condition used as the detail message (for example, "`height < maxHeight`")?**
  While doing so might improve out-of-the-box usefulness of assertions in some cases, the benefit doesn't justify the cost of adding all those string constants to `.class` files and runtime images.

- **Why doesn't an `AssertionError` allow access to the object that generated it? Similarly, why not pass an arbitrary object from the assertion to the `AssertionError` constructor in place of a detail message?**
  Access to these objects would encourage programmers to attempt to recover from assertion failures, which defeats the purpose of the facility.

- **Why not provide context accessors (like `getFile`, `getline`, `getMethod`) on `AssertionError`?**
  This facility is best provided on `Throwable`, so it may be used for all throwables, not just assertion errors. We enhanced `Throwable` with the `getStackTrace` method to provide this functionality.

- **Why is `AssertionError` a subclass of `Error` rather than `RuntimeException`?**

This issue was controversial. The expert group discussed it at length, and came to the conclusion that `Error` was more appropriate to discourage programmers from attempting to recover from assertion failures. It is, in general, difficult or impossible to localize the source of an assertion failure. Such a failure indicates that the program is operating "outside of known space," and attempts to continue execution are likely to be harmful. Further, convention dictates that methods specify most runtime exceptions they may throw (with `@throws` doc comments). It makes little sense to include in a method's specification the circumstances under which it may generate an assertion failure. Such information may be regarded as an implementation detail, which can change from implementation to implementation and release to release.

## Enabling and Disabling Assertions

- **Why not provide a compiler flag to completely eliminate assertions from object files?**

  It is a firm requirement that it be possible to enable assertions in the field, for enhanced serviceability. It would have been possible to also permit developers to eliminate assertions from object files at compile time. Assertions can contain side effects, though they should not, and such a flag could therefore alter the behavior of a program in significant ways. It is viewed as good thing that there is only one semantics associated with each valid Java program. Also, we want to encourage users to leave asserts in object files so they can be enabled in the field. Finally, the spec demands that assertions behave as if enabled when a class runs before it is initialized. It would be impossible to offer these semantics if assertions were stripped from the class file. Note, however, that the standard "conditional compilation idiom" described in the Java Language Specification can be used to achieve this effect for developers who really want it.

- **Why do the commands that enable and disable assertions use package-tree semantics instead of the more traditional package semantics?**

  Hierarchical control is useful, as programmers really do use package hierarchies to organize their code. For example, package-tree semantics allow assertions to be enabled or disabled in all of Swing at one time.

- **Why does `setClassAssertionStatus` return a `boolean` instead of throwing an exception if it is invoked when it's too late to set the assertion status (that is, if the named class has already been initialized)?**

  No action (other than perhaps a warning message) is necessary or desirable if it's too late to set the assertion status. An exception seems unduly heavyweight.

- **Why not overload a single method name to take the place of `setDefaultAssertionStatus` and `setAssertionStatus`?**

  Clarity in method naming is for the greater good. Overloading tends to cause confusion.

- **Why not tweak the semantics of `desiredAssertionStatus` to make it more "programmer friendly" by returning the actual assertion status if a class is already initialized?**

  It's not clear that there would be any use for the resulting method. The method isn't designed for application programmer use, and it seems inadvisable to make it slower and more complex than necessary.

- **Why is there no `RuntimePermission` to prevent applets from enabling/disabling assertions?**

  While applets have no reason to call any of the `ClassLoader` methods for modifying assertion status, allowing them to do so seems harmless. At worst, an applet can mount a weak denial-of-service attack by enabling assertions in classes that have yet to be initialized. Moreover, applets can only affect the assert status of classes that are to be loaded by class loaders that the applets can access. There already exists a `RuntimePermission` to prevent untrusted code from gaining access to class loaders (`getClassLoader`).

- **Why not provide a construct to query the assert status of the containing class?**

  Such a construct would encourage people to inline complex assertion code, which we view as a bad thing. Further, it is straightforward to query the assert status atop the current API, if you feel you must:

```
boolean assertsEnabled = false;
assert assertsEnabled = true;  // Intentional side-effect!!!
// Now assertsEnabled is set to the correct value
```

- **Why does an `assert` statement that executes before its class is initialized behave as if assertions were enabled in the class?**

Few programmers are aware of the fact that a class's constructors and methods can run prior to its initialization. When this happens, it is quite likely that the class's invariants have not yet been established, which can cause serious and subtle bugs. Any assertion that executes in this state is likely to fail, alerting the programmer to the problem. Thus, it is generally helpful to the programmer to execute all assertions encountered while in this state.