[Java 2 SDK for Solaris Developer's Guide](https://docs.oracle.com/cd/E19683-01/806-7930/index.html)  > [Assertion Facility](https://docs.oracle.com/cd/E19683-01/806-7930/6jgp65ikl/index.html)  > [Usage Notes](https://docs.oracle.com/cd/E19683-01/806-7930/6jgp65iks/index.html)  > Preconditions, Postconditions, and Class Invariants

https://docs.oracle.com/cd/E19683-01/806-7930/assert-13/index.html

**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.

**Preconditions**

By convention, preconditions on public methods are enforced by explicit checks inside methods resulting in 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. An assert is inappropriate for such preconditions, as the enclosing method guarantees that it will enforce the argument checks, whether or not assertions are enabled. Further, the assert construct does not throw an exception of the specified type.

If, however, there is a precondition on a nonpublic method and the author of a class believes the precondition to hold no matter what a client does with the class, then an assertion is entirely appropriate. For example:

|  |
| --- |
| /\*\*  \* Sets the refresh interval (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;  ... // Set the refresh interval  } |

Note, the above assertion will fail if MAX\_REFRESH\_RATE is greater than 1000 and the user selects a refresh rate greater than 1000. This would, in fact, indicate a bug in the library!

**Postconditions**

Postcondition checks are best implemented via assertions, whether or not they are specified in public methods. For example:

|  |
| --- |
| /\*\*  \* 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);  if (!this.gcd(m).equals(ONE))  throw new ArithmeticException(this + " not invertible mod " + m);  ... // Do the computation  assert this.multiply(result).mod(m).equals(ONE);  return result;  } |

In practice, one would not check the second precondition (this.gcd(m).equals(ONE)) prior to performing the computation, because it is wasteful. This precondition is checked as a side effect of performing the modular multiplicative inverse computation by standard algorithms.

Occasionally, it is necessary to save some data prior to performing a computation in order to check a postcondition after it is complete. This can be done with two assert statements and the help of a simple inner class designed to save 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 into a functional one:

|  |
| --- |
| void foo(final int[] array) {  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  assert copy.isConsistent();  } |

Note that this idiom easily generalizes to save more than one data field, and to test arbitrarily complex assertions concerning pre-computation and post-computation values.

The first assert statement (which is executed solely for its side-effect) could be replaced by the more expressive:

|  |
| --- |
| copy = new DataCopy(); |

but this would copy the array whether or not asserts were enabled, violating the dictum that asserts should have no cost when disabled.

**Class Invariants**

As noted above, assertions are appropriate for checking internal invariants. The assertion mechanism itself does not enforce any particular style for doing so. It is sometimes convenient to combine many expressions that check required constraints into a single internal method that can then be invoked by assertions. For example, suppose one were to implement a balanced tree data structure of some sort. 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() {  ...  } |

This method is a class invariant. It should always be true before and after any method completes. To check that this is indeed the case, each public method and constructor should contain the line:

|  |
| --- |
| assert balanced(); |

immediately prior to each return. It is generally overkill 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 head of methods in classes whose state is modifiable by other classes. (Better yet, design classes so that their state is not directly visible by other classes!)

© 2010, Oracle Corporation