

- \* An assertion is a statement in the Java™ programming language that enables you to test your assumptions about your program.
- \* 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.
- \* assertions serve to document the inner workings of your program, enhancing maintainability.
- \* `assert Expression1 ;`
  - o where Expression1 is a boolean expression.
  - o When the system runs the assertion, it evaluates Expression1 and if it is false throws an `AssertionError` with no detail message.
- \* `assert Expression1 : Expression2 ;`
  - o Expression1 is a boolean expression.
  - o Expression2 is an expression that has a value.
  - o Expression2 cannot be an invocation of a method that is declared void.
  - o Use this version of the `assert` statement to provide a detail message for the `AssertionError`.
  - o The system passes the value of Expression2 to the appropriate `AssertionError` constructor, which uses the string representation of the value as the error's detail message.
  - o 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.
  - o 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.
  - o 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.
  - o Like all uncaught exceptions, assertion failures are generally labeled in the stack trace with the file and line number from which they were thrown.
  - o 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.
- \* In some cases Expression1 may be expensive to evaluate.
- \* 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.

### Putting Assertions Into Your Code

- \* situations where it is good to use assertions.
  - o Internal Invariants

- o Control-Flow Invariants

- o Preconditions, Postconditions, and Class Invariants

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

- o 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.

- o 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.

```
// Broken! - action is contained in assertion
```

```
assert names.remove(null);
```

```
// Fixed - action precedes assertion
```

```
boolean nullsRemoved = names.remove(null);
```

```
assert nullsRemoved; // Runs whether or not asserts are enabled
```

- o As a rule, the expressions contained in assertions should be free of side effects:

- o evaluating the expression should not affect any state that is visible after the evaluation is complete.

- o One exception to this rule is that assertions can modify state that is used only from within other assertions.

### Internal Invariants

\* 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.

- o The absence of a default case typically indicates that a programmer believes that one of the cases will always be executed.

- o 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.

```
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.

o An acceptable alternative is:

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

o 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 `assert`. (The method would return without a value if no case matched and assertions were disabled.)

### Control-Flow Invariants

\* place an `assert` at any location you assume will not be reached.

\* The `assertions` statement to use is: `assert false;`

```
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.

- o 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.

\* Do not use `asserts` to check the parameters of a public method.

- o An `assert` is inappropriate because the method guarantees that it will always enforce the argument checks.

- o It must check its arguments whether or not `asserts` are enabled.

- o 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.

### Lock-Status Preconditions

\* Classes designed for multithreaded use often have non-public methods with preconditions relating to whether or not some lock is held.

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

\* For example, it is not uncommon to see something like this:

```
// 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
    ...
}
```

### Postconditions

\* You can test postcondition with assertions in both public and nonpublic methods.

```
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.

### 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!)

### Removing all Trace of Assertions from Class Files

- \* strip assertions out of class files entirely.

- o this makes it impossible to enable assertions in the field,

- o it also reduces class file size, possibly leading to improved class loading performance.

- o 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 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

- \* 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:

o no arguments

Enables or disables assertions in all classes except system classes.

o packageName...

Enables or disables assertions in the named package and any subpackages.

o ...

Enables or disables assertions in the unnamed package in the current working directory.

o 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.

\* 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`.

\* 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.

### 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:

o 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.

o 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.

\* `AssertionError` a subclass of `Error` rather than `RuntimeException`

#### Switch Example Meaning

`-ea` Java `-ea` Enable assertions by default

`-da` Java `-da` Disable assertions by default

`-ea:<classname>` Java `-ea:AssertPackageTest` Enable assertions in class `AssertPackageTest`

`-da:<classname>` Java `-da:AssertPackageTest` Disable assertions in class `AssertPackageTest`

`-ea:<packagename>...` - Java `-ea:pkg0...` Enable assertions in package `pkg0`

`-da:<packagename>...` Java `-da:pkg0...` Disable assertions in package `pkg0`

`-esa` Java `-esa` Enable assertions in system classes

`-dsa` Java `-dsa` Disable assertions in system classes