Programming with assertions
Oracle guidelines

J.Serrat

102759 Software Design

October 27, 2015
1 Preliminaries

2 When *not* to use assertions

3 When to use assertions

4 Post-conditions

5 Requiring enabled
**Assertions**

Why assertions?

*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.

In Java, the assert construct is not a full-blown design-by-contract facility like in Eiffel or other languages. But it supports an *informal* D-by-C programming style.
Assertions

Syntax

```markdown
assert boolean-expression [: value-expression]
```

- if boolean expression evaluates to `false` throws an `AssertionError` exception
- if not caught, ultimately execution stops
- `value-expression` normally a string message plus some data: details of assertion failure
- message not intended to end-users but programmers
class Interval {
    public Interval(Task t) {
        assert (t != null) : "Null parent task";
        //...
    }
}

Exception in thread "main" java.lang.AssertionError: Null parent task at nucli.Task(Task.java:13)
By default, disabled: they’re just comments.

To enable all assertions: in Eclipse IDE Run → Run configurations... → (x)=Arguments → VM arguments → write -ea or -enableassertions

To disable assertions: -da or -disableassertions

You can enable and disable assertions at package and class level: -ea:nucli -da:nucli.Client
When **not** to use assertions

For argument checking in public constructors: better throw a proper runtime exception: `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`

```java
class Interval {
    public Interval(Task t) {
        throw new IllegalArgumentException("Null parent task in Interval constructor");
    }
}
```

Exception in thread "main"
java.lang.IllegalArgumentException: Null parent task in Interval constructor at nucli.Task(Task.java:13)
When **not** to use assertions

When the evaluation of boolean expression contains an action required for correct operation: assertions may be enabled *or not* ⇒ action may perform or not.

```java
assert (name = name.trim()).length() >0 : "blank name";
```
When to use assertions

In case we come across an **internal invariant**: something which is always true, but not a class invariant.

```java
class Unlock extends CodeHandler {
    @Override
    public void handleCode(String inputCode, Door door) {
        if (doorState.isLocked()) {
            if (code.equals(inputCode)) {
                doorState.reset();
                // end of chain
            } else {
                // any other code is invalid
                logger.info("Door\{\} is not the unlock code", door.getId(), inputCode);
                // end of chain
            }
        } else if (next != null) {
            // it should be, to reach Lock
            logger.debug("pass input code from Unlock to next");
            next.handleCode(inputCode, door);
        }
    }
}
```
**When to use assertions**

Better than a comment, use assertion:

```java
class Unlock extends CodeHandler {
    @Override
    public void handleCode(String inputCode, Door door) {
        if (doorState.isLocked()) {
            if (code.equals(inputCode)) {
                doorState.reset();
                // end of chain
            } else {
                // any other code is invalid
                logger.info("Door \{\} \{-\\} is not the unlock\" + "code", door.getId(), inputCode);
                // end of chain
            }
        } else {
            assert (next != null) : "Unlock has not next" + "handler";
            next.handleCode(inputCode, door);
        }
    }
}
```
When to use assertions

switch statement with no default case:

```java
switch (direction) {
    case Directions.North:
        //...
        break;
    case Directions.East:
        //...
        break;
    case Directions.South:
        //...
        break;
    case Directions.West:
        //...
        break;
    default:
        assert false : "Unknown direction" + direction
        // alternatively, works even if assertions not enabled
        throw new AssertionError("Unknown direction" + direction);
}
```
### When to use assertions

At places where execution thread is not supposed to reach

```java
void computation() {
    for (...) {
        if (condition) {
            // condition is always true sometime
            return;
        }
    }
    assert false : "Execution should never reach here";
}
```
When to use assertions

In pre-conditions of public methods don’t use `assert`. We want public methods always check arguments, and throw an exception of some pre-defined type.

```java
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);
}

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
}
```
Post-conditions are bothersome to implement because we may need to keep a copy of the value of attributes and parameters at the beginning of a method, to compare to their value before exiting.

```java
void foo(int[] array) {
    ...
    // want to check that the array has not changed in elements
    // or their order
    return;
}
```
You can do it with an inner class:

```java
void foo(final int[] array) {
    // 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);
    //...
    assert copy.isConsistent();
}
```
How to avoid disabling assertion checking in a critical class? Put this static initialization *idiom* at the top of your class:

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

Prevents a class from being initialized if its assertions have been disabled.