Java: Assertion

IF2210 – Semester II 2020/2021

by: SA, IL, Yohanes Nugroho; rev: SAR

Bahan diekstrak dari:
http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html
http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

Assertion

- 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.

Assertion Statement

- 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 Expression1: Expression2;
```

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

Penulisan Assertion

- Asersi bahwa `speed` pasti `< 300000`:

```
assert (speed<300000);
```

- Asersi juga bisa diberi pesan yang lebih informatif:

```
assert (speed<300000): "kecepatan melebihi kecepatan
cahaya, menembus waktu";
```

Pengaktifan Asersi

- Defaultnya assersi tidak akan diperiksa oleh JVM.
- Asersi diaktifkan (diperiksa) jika program dijalankan dengan *option* `-ea` (*enable assertion*):

```
java -ea Speed
```

Assert in C++ (1)

- Pada C++, assertion diimplementasi sebagai Macro `<cassert>`

```
void assert (int expression);
```

- Evaluate assertion
  - If the argument expression of this macro with functional form compares equal to zero (i.e., the expression is false), a message is written to the standard error device and abort is called, terminating the program execution.
  - The specifics of the message shown depend on the particular library implementation, but it shall at least include: the expression whose assertion failed, the name of the source file, and the line number where it happened. A usual expression format is:

    Assertion failed: expression, file filename, line line number

Assert in C++ (2)

- This macro is disabled if, at the moment of including `<assert.h>`, a macro with the name `NDEBUG` has already been defined.
- This allows for a coder to include as many assert calls as needed in a source code while debugging the program and then disable all of them for the production version by simply including a line like:

```
#define NDEBUG
```

  at the beginning of its code, before the inclusion of `<assert.h>`.

- Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

```cpp
using namespace std;

#include <iostream>
#include <assert.h>     /* assert */

void print_number(int x) {
  assert (x>0);
  cout << x << endl;
}

int main () {
  int a=10;   int b = 0;  int c = -6;
  print_number(1);
  print_number(b);
  print_number(c);
  return 0;
}
```

```
Output:
1
a: asersi.cpp:14: void print_number(int): Assertion `x>0' failed.
Aborted
```

Assertion

- Assertion
  - *Statement* yang memfasilitasi programmer untuk menguji bahwa asumsinya terhadap program berlaku dengan baik.
  - Direpresentasikan dalam ekspresi `boolean` yang dianggap benar jika dieksekusi. Jika tidak, sistem akan melakukan *throw error*.
- JAVA :

```
assert Expression1; // jika gagal system akan
// throw sebuah AssertionError tanpa pesan detil

assert Expression1 : Expression2; // ekpresi ke-2
// adalah ekspresi yang menghasilkan nilai,
// untuk pesan kesalahan bagi programmer
```

Kapan Assertion Digunakan?

- Assertion sebaiknya digunakan pada situasi:
  - Internal Invariants
  - Control-Flow Invariants
  - Preconditions, Postconditions, and Class Invariants
- Assertion sebaiknya TIDAK digunakan untuk:
  - Cek argumen method yang bersifat publik
    - karena argumen tsb mrp bagian dari kontrak
  - Melakukan aksi/operasi yang benar yang dilakukan oleh program
    - karena assertion bisa di-disable

Internal Invariants [1]

- http://docs.oracle.com/cd/E19683-01/806-7930/assert-11/index.html
- In general, it is appropriate to frequently use short assertions indicating important assumptions concerning a program's behavior.
- In the absence of an assertion facility, many programmers use comments in the following way:

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

Internal Invariant [2]

- When your code contains a construct that asserts an invariant, you should change it to an assert. To change the above example (where an assert protects the else clause in a multiway if-statement), you might do the following:

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

Control-Flow Invariants

- Another good candidate for an assertion is a switch statement with no default case.

```
switch(suit) {
    case Suit.CLUBS:
        ...
        break;
    case Suit.DIAMONDS:
        ...
        break;
    case Suit.HEARTS:
        ...
        break;
    case Suit.SPADES:
        ...
}
```

Control Flow

- The programmer probably assumes that one of the four cases in the above switch statement will always be executed. To test this assumption, add the following default case:

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

        ...
        break;
    default:
        assert false;
}
```

Preconditions

- Anda sudah mengenal Prekondisi/*Initial state* pada saat mendefinisikan fungsi/prosedur.
- Kontrak bahwa pemakai fungsi/prosedur harus menaatinya.
- Selama ini hanya dituliskan sebagai komentar.
- Dapat di-*enforce* secara eksplisit dengan pemeriksaan kasus (if ... then throw ...) atau dituliskan sebagai asersi.

Contoh Prekondisi

- 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
}
```

Post conditions

- Anda sudah mengenal *Post condition/Final state* pada saat mendefinisikan fungsi/prosedur.
- Kontrak bahwa penyedia fungsi/prosedur harus menaatinya.
- Selama ini hanya dituliskan sebagai komentar.
- Sebaiknya dituliskan sebagai asersi.

- 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;
}
```

Class Invariants (1)

- 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.

Class Invariants (2)

- Menjamin bahwa properti suatu kelas benar.
- Misalnya untuk membentuk sebuah JAM, harus dijamin nilai Hour [0..23], Minutes [0..59], Second [0..59]
- ctor(h,m,s : integer) hanya dapat menciptakan dengan benar jika parameter yang disuplai benar
- Jika anda mendefinisikan sebagai integer, maka harus dituliskan sebagai asersi

Bad Use Of Assertion (1)

http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

- **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.

Bad Use Of Assertion (2)

- **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;
// remove() 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.