Warunki poprawności

Czym są asercje?

Asercja to fragment kodu sprawdzający wewnętrzny stan programu i przerywający jego działanie w przypadku błędu.

Po co dodatkowy mechanizm do wykrywania błędów skoro mamy wyjątki? Wyjątki są wykorzystywane do obsługi sytuacji mało prawdopodobnych, lecz możliwych i koniecznych do obsłużenia (np. brak pamięci, zerwane połączenie sieciowe, błędny format liczby, dzielenie przez zero, itp.).

Asercje natomiast definiują warunki poprawności, które zawsze muszą być poprawne (np. wynik funkcji obliczającej długość odcinka musi być większa od 0, parametr zawierający pole kwadratu musi być dodatnie, funkcja sortująca zwraca posortowane dane, itp.). Jeżeli te warunki będą fałszywe, to ewidentnie występuje błąd w programie.

Kiedy i gdzie korzystać z asercji

Krótka odpowiedź brzmi: wszędzie. Gdziekolwiek przypuszczasz, że stan programu powinien być taki, jakiego oczekujesz, lecz nie jest to oczywiste z poprzedzających instrukcji powinno się użyć asercji, aby to sprawdzić. Najważniejsze miejsca:

- Na początku każdej procedury (aby sprawdzić poprawność parametrów wejściowych)
- Na końcu każdej procedury (aby sprawdzić poprawność generowanych wyników)
- Na początku każdej pętli (aby sprawdzić niezmienniki pętli)
- Na końcu każdej pętli (sprawdzenie wyników działania pętli)
- Przed użyciem zmiennej (sprawdzenie, czy nie jest równa null)

Z asercji możemy korzystać w dwojaki sposób:

- Sprawdzanie poprawności stanu przed i po wykonaniu operacji
- Duplikowanie kodu (np. napisanie skomplikowanego algorytmu w prostszy sposób), uruchomienie dwóch fragmentów i porównanie wyników (np. testowanie algorytmu heapsort z wykorzystaniem sortowania bąbelkowego)

Asercje w Javie

Instrukcja *assert* przyjmuje dwie formy. Pierwsza prostsza:

```
assert Expression1 ;
```

gdzie $Expression_1$ jest wyrażeniem logicznym. W momencie uruchomienia asercji, wyrażenie $Expression_1$ jest obliczane w przypadku wartości false jest rzucany wyjątek `AssertionError` bez szczegółowego komunikatu.

Druga postać:

```
assert Expression1 : Expression2 ;
```

gdzie:

- *Expression*$_1$ jest wyrażeniem logicznym.
- *Expression*$_2$ zawiera komunikat szczegółowy, który zostanie przekazany do wyjątku i wyświetlony na ekranie.

Zadanie

Zadaniem jest stworzenie klasy StrangeSet, której szkielet wraz z komentarzem wygląda następująco:

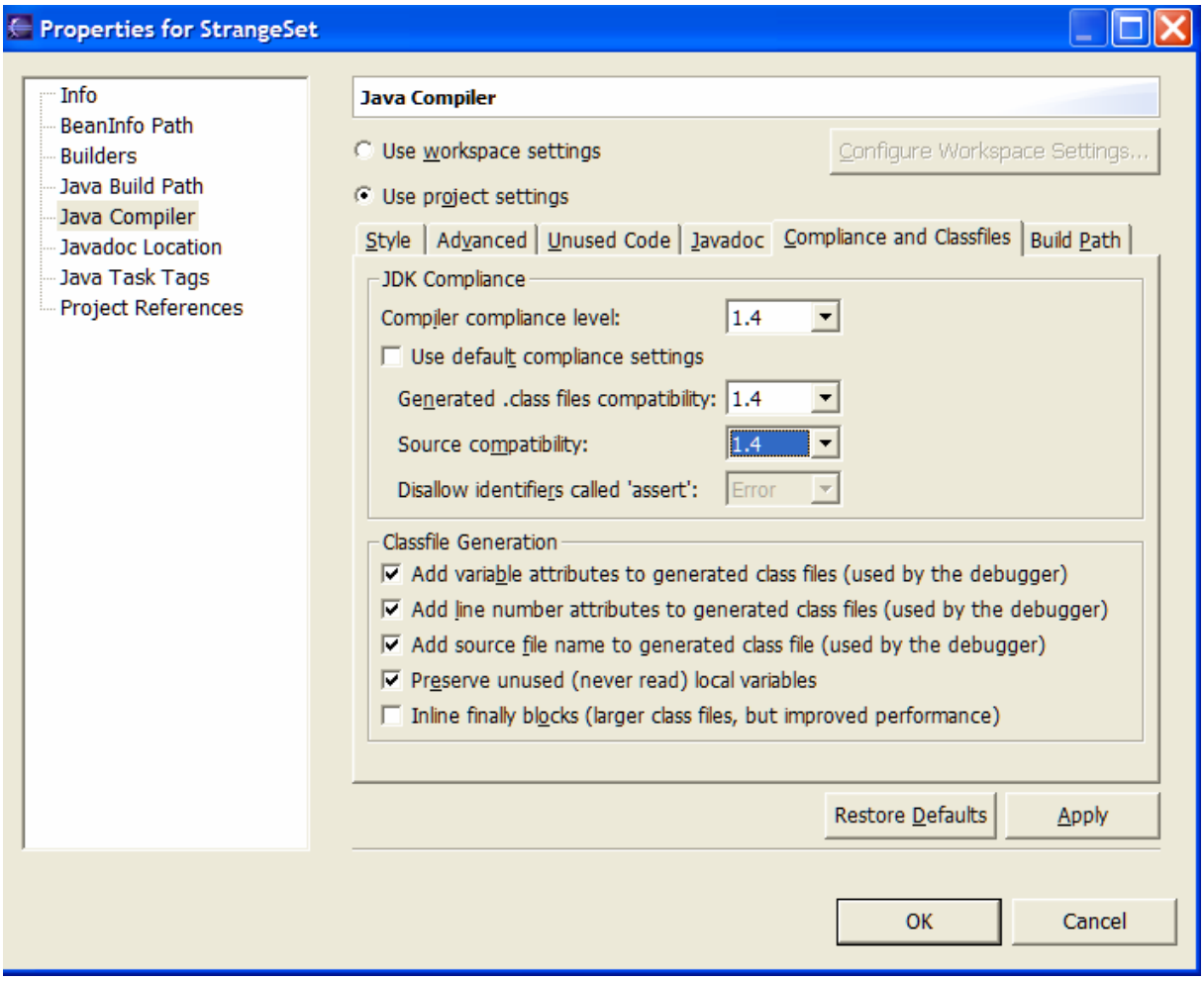
```
package pl.put.cs.iolab.assertions;


public class StrangeSet {

        public static int MAX_SIZE = 1000;
        protected int[] set = new int[MAX_SIZE];
        protected int size;

        /**
         * Dodaje liczbę k do zbioru liczb.
         * Jeżeli podana liczba już istnieje dodawana jest po raz drugi
         * @param k liczba, którą należy dodać od zbioru
         * @throws Exception występuje w przypadku przepełnienia tablicy
         */
        public void add(int k) throws Exception{
        }

        /**
         * Usuwa liczbę k ze zbioru liczb.
         * W przypadku gdy zbiór nie posiada liczby podanej jako parametr rzucany jest wyjątek.
         * @param k liczba do usunięcia
         * @throws Exception w przypadku gdy zbiór nie posiada danej liczby
         */
        public void remove(int k) throws Exception{
        }

        /**
         * Losuje jedną liczbę ze zbioru liczb oraz usuwa ją ze zbioru.
         * @return wylosowana liczba
         * @throws Exception występuje w przypadku pustego zbioru
         */
        public int drawAtRandom() throws Exception{
                return 0;
        }
```

```
	/**
	 * Zwraca sumę wszystkich liczb ze zbioru.
	 * @return Suma liczb. W przypadku pustego zbioru wynosi 0.
	 */
	public int getSumOfElements() {
		return 0;
	}

	/**
	 * Dzieli każdy element ze zbioru przez n bez reszty.
	 * @param n liczba przez którą będzie wykonane dzielenie.
	 */
	public void divideAllElementsBy(int n) {
	}

	/**
	 * Sprawdza, czy w zbiorze istnieje element k
	 * @param k element do sprawdzenia
	 * @return true w przypadku odnalezienia elementu, false w przeciwnym przypadku
	 */
	public boolean contains(int k) {
		return false;
	}

	/**
	 * Zwraca rozmiar zbioru, czyli faktycznie dodanych liczbę elementów
	 * @return rozmiar zbioru
	 */
	public int getSize() {
		return 0;
	}

}
```

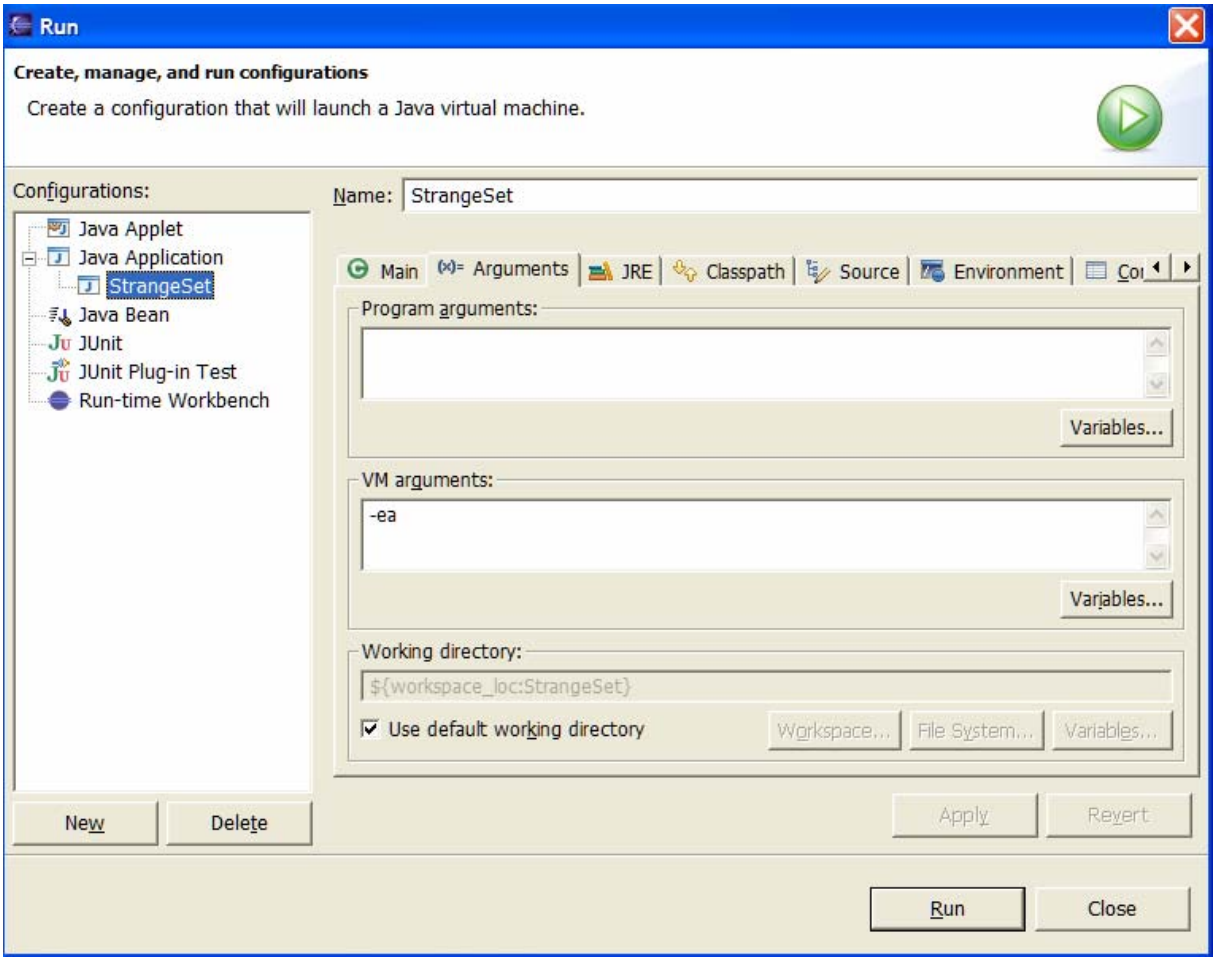
Do każdej funkcji należy wymyślić po kilka asercji, które będą sprawdzać poprawność programu. Następnie stworzyć klasę testującą, która będzie wykonywać pewne operacje na zbiorze liczb (w ten sposób kod asercji będzie uruchamiany).

Asercje w środowisku Eclipse

W środowisku Eclipse asercje są domyślnie wyłączone. Aby je włączyć należy wykonać dwie rzeczy.
Po pierwsze ustawić właściwości kompilatora, aby wygenerował kod asercji. Należy w tym celu wybrać opcję *Project->Properties->Java Compiler*, a następnie ustawić opcje na zakładce *Compliance and Classfiles* zgodnie z poniższym rysunkiem:



Następnie trzeba uruchomić program: *Run->Run As->Java Application* – zostanie stworzona domyślna konfiguracja uruchamiania, którą zmodyfikujemy aby uruchomić asercje.
W tym celu wybieramy *Run->Run…*, a następnie szukamy konfiguracji *StrangeSet* pod *Java Application* i na zakładce *Arguments* wpisujemy „-ea” (skrót od *Enable Assertions*) w polu *VM arguments:*



Po uruchomieniu programu („Run”) komunikaty o asercjach będą się pojawiały w widoku konsoli (*Window->Show view->Console*).

Uwaga: jeżeli chcemy ponownie uruchomić program, który uruchamialiśmy ostatnio wystarczy nacisnąć przycisk  na pasku narzędzi – domyślnie zostanie wybrana ostatnia konfiguracja uruchamiania.

Fragment artykułu: „Programming with Assertions”

http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html

There are many situations where it is good to use assertions. This section covers some of them:

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

There are also a few 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 (JLS 14.20), 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 invariants 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!)