Assertions, pre/post-conditions and invariants

Programming as a contract

- Specifying what each method does
  - Specify it in a comment before method's header
- Precondition
  - What is assumed to be true before the method is executed
  - **Caller obligation**
- Postcondition
  - Specifies what will happen if the preconditions are met
  - **Method obligation**

Class Invariants

- A **class invariant** is a condition that all objects of that class must satisfy while it can be observed by clients

What is an assertion?

- An *assertion* is a statement that says something about the state of your program
- Should be true if there are no mistakes in the program

```
//n == 1
while (n < limit) {
  n = 2 * n;
}
// what could you state here?
```

What is an assertion?

- An *assertion* is a statement that says something about the state of your program
- Should be true if there are no mistakes in the program

```
//n == 1
while (n < limit) {
  n = 2 * n;
}
//n >= limit
//more?
```

What is an assertion?

- An *assertion* is a statement that says something about the state of your program
- Should be true if there are no mistakes in the program

```
//n == 1
while (n < limit) {
  n = 2 * n;
}
//n >= limit
//n is the smallest power of 2 >= limit
```

assert

Using **assert**:

```
assert n == 1;
while (n < limit) {
    n = 2 * n;
}
assert n >= limit;
//n is the smallest power of 2 >= limit.
```

When to use Assertions

- We can use assertions to guarantee the behavior.

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

int p=..,d=..,r,q;
q = p/d;
r = p%d;
assert  ??
```

Control Flow

- If a program should never reach a point, then a constant false assertion may be used

```
void search() {
    for (...) {
        ...
        if (found) // will always happen
            return;
    }
    assert false; // should never get here
}
```

Assertions

- Syntax:

```
assert Boolean_Expression;
```

- Each assertion is a boolean expression that you claim is true.
- By verifying that the boolean expression is indeed true, the assertion confirms your claims about the behavior of your program, increasing your confidence that the program is free of errors.
- If assertion is false when checked, the program terminates and an error message is printed.

When to use assertions?

- Programming by contract
- **Preconditions** in methods (eg value ranges of parameters) should be enforced rather than asserted
- **Postconditions**
  - Assert post-condition

Performance

- Assertions may slow down execution. For example, if an assertion checks to see if the element to be returned is the smallest element in the list, then the assertion would have to do the same amount of work that the method would have to do
- Therefore assertions can be **enabled** and **disabled**
- Assertions are, by default, disabled at run-time
- In this case, the assertion has the same semantics as an empty statement
- Think of assertions as a debugging tool
- Don’t use assertions to flag user errors, because assertions can be turned off

Assertions in Eclipse

- Go to Preferences -> Java -> Compiler and set the Compiler Compliance Level to 1.5 or 1.6. Also check Use Default compliance settings. This tells the compiler to recognize and allow assert statements, but does not enable them.
- To enable assert statements, you must set a compiler flag. Go to Run -> Run Configurations -> Arguments, and in the box labeled VM arguments, enter either -enableassertions or just -ea

More Information

- For more information:

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

Loop invariants

- We can use **predicates** (logical expressions) to reason about our programs.
- A **loop invariant** is a predicate
  - that is true directly before the loop executes
  - that is true before and after the loop body executes
  - and that is true directly after the loop has executed

  Ie, it is kept invariant by the loop.

Loop invariants cont'

- Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop:

```
 <loop invariant>
while(test){
    <loop invariant>
     S;
    <loop invariant>
}
< not test AND loop invariant>
```

What does it mean...

```
<loop invariant>
while(test){
    <loop
   invariant>
       S;
    <loop
   invariant>
}
< not test AND
 loop invariant>
```

If we can prove that

. the loop invariant holds before the loop

and that

. the loop body keeps the loop invariant true
  ie. <test AND loop invariant> S; <loop invariant>

then we can infer that

. not test AND loop invariant
  holds after the loop terminates

Example: loop index value after loop

```
<precondition: n>0>
int i = 0;
 while (i < n){
    i = i+1;
}
<post condition: i==n >
```

We want to prove:
i==n right after the loop

Example: loop index value after loop

```
<precondition: n>0>
int i = 0;
// i<=n    loop invariant
 while (i < n){
    // i < n  test passed
    //   AND
    //   i<=n  loop invariant
       i++;
    // i <= n  loop invariant
}
// i>=n  AND i <= n  → i==n
```

So we can conclude the obvious:

i==n right after the loop

Example summing

```
int total (int[] elements){
   int sum = 0,i = 0, n = elements.length;
   // sum has sum of elements from 0 to i-1      the empty set
    while (i < n){
       // sum == sum of elements 0..i-1
        sum  += elements [i];
        i++;
       // sum == sum of elements 0..i-1
   }
   // i==n  (previous example)  AND
   // sum has sum elements 0..i-1  →  sum == sum of elements 0..n-1
   //                                 → sum == sum of int[] elements
 return sum;
}
```

Summary: Loop Invariant Reasoning

```
//loop invariant true before loop
while (b){
   // b  AND loop invariant
       S;
   // loop invariant
}
// not b  AND loop invariant
```

not b helps you make a stronger observation than loop invariant alone.