Assertions, pre/post-conditions and invariants

Assertions: Section 4.2 in Savitch (p. 239)
Loop invariants: Section 4.5 in Rosen

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 – what the method guarantees to the caller
  - **Method obligation**

Example

```
/*
 ** precondition:  x >= 0
 ** postcondition: return value satisfies:
 ** result * result == x
*/
double sqrt(double x) {

}
```

Enforcing preconditions

```
/*
 ** precondition:  x >= 0
 ** postcondition: return value satisfies:
 ** result * result == x
*/
double sqrt(double x) {
  if (x < 0)
     throw new ArithmeticExpression ("you
       tried to take sqrt of a neg number!");

}
```

Class Invariants

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

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
```

assert

Using `assert:`

```
assert n == 1;
while (n < limit) {
    n = 2 * n;
}
assert n >= limit;
```

When to use Assertions

- Another example

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

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; ... }
```

Another example

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

Another example

```
int p=…,d=…;
int q = p/d;
int r = p%d;
assert p == q*d + r;
```

Control Flow

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

```
private 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 raises an exception.

When to use assertions?

- Programming by contract
- **Preconditions** in methods (eg value ranges of parameters) should be enforced rather than asserted because assertions can be turned off
- **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

- 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

  i.e., it is kept invariant by the loop.

Loop invariants

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

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

What does it mean...

```
<loop invariant>
while(test){
   <test AND
   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
i.e. <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: sum of elements in an array

```
int total (int[] elements){
   int sum = 0, i = 0, n = elements.length;
   // sum == sum of elements from 0 to i-1
   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 == sum elements 0...i-1
   // →  sum == sum of elements 0...n-1
   return sum;
}
```

Closed Curve Game

- There are two players, Red and Blue. The game is played on a rectangular grid of points:

```
6   . . . . . . .
5   . . . . . . .
4   . . . . . . .
3   . . . . . . .
2   . . . . . . .
1   . . . . . . .
    1 2 3 4 5 6 7
```

Red draws a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by a line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red's goal is to form a closed curve of red line segments. Blue's goal is to prevent Red from doing so.

See http://www.cs.uofs.edu/~mccloske/courses/cmps144/invariants_lec.html

Closed Curve Game

- We can express this game as a computer program:

```
while (more line segments can be drawn) {
      Red draws line segment;
      Blue draws line segment;
 }
```

Question: Does either Red or Blue have a winning strategy?

Closed Curve Game

- **Answer:** Yes! Blue is guaranteed to win the game by responding to each turn by Red in the following manner:

```
if (Red drew a horizontal line segment) {
    let i and j be such that Red's line segment connects (i,j) with (i,j+1)
    if (i>1) {
       draw a vertical line segment connecting (i-1,j+1) with (i,j+1)
    } else {
       draw a line segment anywhere
    }
 } else  // Red drew a vertical line segment
    let i and j be such that Red's line segment connects (i,j) with (i+1,j)
    if (j>1) {
       draw a horizontal line segment connecting (i+1,j-1) with (i+1,j)
    } else  {
       draw a line segment anywhere
    }
 }
```

Closed Curve Game

- By following this strategy Blue guarantees that Red does not have an “upper right corner” at any step.
- So, the invariant is:

  There does not exist on the grid a pair of red line segments that form an upper right corner.

  And in particular, Red has no closed curve!

Example: Egyptian multiplication

		A	B	
		19	5	
19 x 5:	/2	9	10	*2
	/2	4	20	*2
	/2	2	40	*2
	/2	1	80	*2

throw away all rows with even A:

A	B
19	5
9	10
1	80

add B's 95

--> the product !!

Can we show it works? Loop invariants!!

```
// pre: left >0 AND right >0
int a=left, b=right, p=0;  //p:  the product
// p + (a*b) == left * right   loop invariant
while (a!=0){
   // a!=0 and p + (a*b) == left * right
   // loop condition and loop invariant
   if (odd(a))  p+=b;
   a/=2;
   b*=2;
  // p+(a*b) == left*right
}
// a==0 and p+a*b == left*right  →  p == left*right
```

Try it on 7 * 8

left	right	a	b	p
7	8	7	8	0
		3	16	+=b: 8
		1	32	+=b: 24
		0	64	+=b: 56

Try it on 8*7

left	right	a	b	p
8	7	8	7	0
		4	14	0
		2	28	0
		1	56	0
		0	118	+=b: 56

Relation to int representation 19*5

```
  00101
  10011
 ______
    101   5
   1010  10
  00000
 000000
1010000  80
_______
1011111  95
```

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.