Assertions, pre/post-conditions

Assertions: Section 4.2 in Savitch (p. 239)
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

```c
/*
 ** 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 ArithmeticException("you tried to take sqrt of a neg number!");
}
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.

```java
//n == 1
while (n < limit) {
    n = 2 * n;
}
// what will be the 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.

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

//can you make that stronger?
```
Using `assert`:

```plaintext
assert n == 1;
while (n < limit) {
    n = 2 * n;
}
assert n >= limit;
```
When to use Assertions

- Another example

```java
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.

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

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

```c
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

```java
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.
Asserts in switch statements

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

If your program is correct, one of these cases should hold!

How to use assertions to verify that?
Assertions in switch statements

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

Assertions in switch statements

switch(suit) {
    case Suit.CLUBS:
        ...
        break;
    case Suit.DIAMONDS:
        ...
        break;
    case Suit.HEARTS:
        ...
        break;
    case Suit.SPADES:
        ...
        break;
    default:
        assert false : suit;
        //gives the value that violated the assertion
}
Assertions

Let’s take a closer look at the assertion statement:

```java
assert false : suit;
```

This uses a more general form of the assert statement:

```java
assert Expression₁ : Expression₂;
```

- `Expression₁` is a boolean expression.
- `Expression₂` is an expression that has a value. (It cannot be an invocation of a method that is declared void.)
- Use this version of the assert statement to provide a message with the AssertionError. The system passes the value of `Expression₂` to the appropriate AssertionError constructor.
When to use assertions?

- Programming by contract
- **Preconditions** in methods (e.g., value ranges of parameters) should be enforced rather than asserted because assertions can be turned off
- **Postconditions**
  - Assert post-condition
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
Class Invariants

- A class invariant is a condition that all objects of that class must satisfy while it can be observed by clients.
- Example: your bank balance should always be positive.
- Verify a class invariant using assertions.
Loop invariants

- We 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 therefore true directly after the loop has executed

  i.e., it is kept invariant by the loop.
Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop:

\[<\text{loop invariant}>\]
\[\text{while}(\text{test})\{\]
\[<\text{test AND loop invariant}>\]
\[S;\]
\[<\text{loop invariant}>\]
\[\}
\[<\text{not test AND loop invariant}>\]
What does it mean...

\[
\text{<loop invariant> while(test)\{} \text{<test AND loop invariant>} S; \text{<loop invariant>} \}\text{< not test AND loop invariant>}
\]

\text{If we can prove that}
\[
\begin{align*}
\text{. & the loop invariant holds before the loop} \\
\text{and that} \\
\text{. & the loop body keeps the loop invariant true} \\
\text{i.e. <test AND loop invariant> S; <loop invariant>}
\end{align*}
\]

\text{then we can infer that}
\[
\begin{align*}
\text{. not test AND loop invariant} \\
\text{holds after the loop terminates}
\end{align*}
\]
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  \(\rightarrow\) i==n

So we can conclude the obvious:

\(i==n\) right after the loop
Example summing

```java
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 \rightarrow sum == sum of elements 0..n-1
    // \rightarrow sum == sum of int[] elements
    return sum;
}
```
Example: Egyptian multiplication

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>5</td>
</tr>
</tbody>
</table>

19 \times 5:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>/2</td>
<td>9</td>
<td>10</td>
<td>*2</td>
</tr>
<tr>
<td>/2</td>
<td>4</td>
<td>20</td>
<td>*2</td>
</tr>
<tr>
<td>/2</td>
<td>2</td>
<td>40</td>
<td>*2</td>
</tr>
<tr>
<td>/2</td>
<td>1</td>
<td>80</td>
<td>*2</td>
</tr>
</tbody>
</table>

Throw away all rows with even A:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
</tr>
</tbody>
</table>

\[
\text{add B's} \quad 95
\]

\[
\text{--> the product} \quad \text{!!}
\]
Can we show it works? Loop invariants!!

// pre:   left >=0 AND right >=0
int a=left, b=right, p=0;
// 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

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>a</th>
<th>b</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>+b: 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>+b: 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>+b: 56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>right</td>
<td>a</td>
<td>b</td>
<td>p</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>56</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>118</td>
<td>+=b: 56</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Relation to int representation $19*5$

00101
10011

_____
101 5
1010 10
00000
000000
000000
1010000 80

_____
1011111 95 = 64 + 31
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.
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.