Loop Invariants: Part 1
Reasoning About Method Calls

• What a method call does is described by its *contract*
  – Precondition: a property that is true *before* the call is made
  – Postcondition: a property that is true *after* the call returns
Reasoning About Loops

• What a **while** loop does is described by its **loop invariant**
  – Invariant: a property that is true *every time* the code reaches a certain point—in the case of a loop invariant, the loop condition test
Reasoning About Loops

• What a **while** loop does is described by its **loop invariant**
  - Invariant: a property that is true *every time* the code reaches a certain point—in the case of a loop invariant, the loop condition test

Why is a loop treated differently than a method call? Simply put, experience shows this is a good way to *think about* loops.
Reasoning About Loops

- What a **while** loop does is described by its **loop invariant**
  - Invariant: a property that is true *every time* the code reaches a certain point—in the case of a loop invariant, the loop condition test

Just **while** loops? Yes; the same idea can be applied to **for** loops, but some modifications are required.
Reasoning About Loops

• What a **while** loop does is described by its **loop invariant**
  – Invariant: a property that is true *every time* the code reaches a certain point—in the case of a loop invariant, the loop condition test

  Since a loop invariant is true *every time* through the loop, it says what **does not change**; hence it really says what the loop **does not do**.
while Statement Control Flow

while (test) {
  loop-body
}

---

while (test) {
  loop-body
}
while (test) {
    loop-body
}

The loop invariant is a property that is true both here, just before the loop begins...
while (test) {
    loop-body
}

... and here, just after every execution of the loop body.
Example #1

```cpp
void append(Queue<T> q)

• Concatenates (“appends”) q to the end of this.

• Updates: this

• Clears: q

• Ensures:

  this = #this * #q
```
Example #1: Method Body

```java
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
```
Example #1: Method Body

```java
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
```

What is true every time we test the loop condition? Lots of things... such as?
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}

this = < 1, 2, 3 >
q = < 4, 5, 6 >
This is the first time we test the loop condition.

Lots of things... such as?
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}

this = <1, 2, 3>
q = <4, 5, 6>

T x = q.dequeue();
this = <1, 2, 3>
q = <4, 5, 6>
x = 4

this.enqueue(x);
this = <1, 2, 3, 4>
q = <5, 6>
x = 4


```java
T x = q.dequeue();
this.enqueue(x);
```

What is true the first and second times we test the loop condition? Fewer things... such as?

<table>
<thead>
<tr>
<th>First Test</th>
<th>Second Test</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this = &lt;1, 2, 3&gt;</code></td>
<td><code>this = &lt;1, 2, 3, 4&gt;</code></td>
</tr>
<tr>
<td><code>q = &lt;4, 5, 6&gt;</code></td>
<td><code>q = &lt;5, 6&gt;</code></td>
</tr>
<tr>
<td><code>x = 4</code></td>
<td><code>x = 4</code></td>
</tr>
</tbody>
</table>
The value of $x$ is not involved in the loop invariant because there is no $x$ when we first hit the loop!
```java
this = < 1, 2, 3 >
q = < 4, 5, 6 >

while (q.length() > 0) {
    this = < 1, 2, 3, 4 >
    q = < 5, 6 >
    T x = q.dequeue();
    this = < 1, 2, 3, 4 >
    q = < 6 >
    x = 5
    this.enqueue(x);
    this = < 1, 2, 3, 4, 5 >
    q = < 6 >
    x = 5
}
```
```java
while (q.length() > 0) {
    T x = q.dequeue();
    this = <1, 2, 3, 4>
    q = <5, 6>
    x = 5
    this.enqueue(x);
    this = <1, 2, 3, 4, 5>
    q = <6>
    x = 5
}
```

What is true the first, second, and third times we test the loop condition? Fewer things still... such as?
```java
this = < 1, 2, 3 >
q = < 4, 5, 6 >

while (q.length() > 0) {

this = < 1, 2, 3, 4, 5 >
q = < 6 >

T x = q.dequeue();

this = < 1, 2, 3, 4, 5 >
q = < >
x = 6

this.enqueue(x);

this = < 1, 2, 3, 4, 5, 6 >
q = < >
x = 6
}
```
```java
while (q.length() > 0) {
    T x = q.dequeue();
    this = < 1, 2, 3, 4, 5 >
    q = < >
    x = 6
    this.enqueue(x);
    this = < 1, 2, 3, 4, 5, 6 >
    q = < >
    x = 6
}
```

What is true the first, second, third, and fourth times we test the loop condition? Fewer things still... such as?
| while (q.length() > 0) { | \texttt{this} = \langle 1, 2, 3 \rangle  \\
| \text{\texttt{this} = \langle 1, 2, 3, 4, 5 \rangle} | q = \langle 4, 5, 6 \rangle \\
| T x = q.dequeue(); | this = \langle 1, 2, 3, 4, 5 \rangle  \\
| this.enqueue(x); | x = 6  \\
| } | this = \langle 1, 2, 3, 4, 5, 6 \rangle  \\
| q = \langle \rangle | x = 6  \\

Whatever is true the last time we test the loop condition is also true here, after the loop finally terminates.
Some Things That Do Not Change

• “The lengths of the strings are non-negative” does not change
  – $|\textit{this}| \geq 0 \text{ and } |q| \geq 0$ does not change
  – True, but this literally goes without saying; the length of any string is always non-negative
  – It is no more useful than saying, e.g., “17 < 42 does not change”, because it is a mathematical fact, not something about this loop in particular
Some Things That Do Not Change

• “The sum of the lengths of the strings” does not change
  – $|this| + |q|$ does not change
  – True, and a useful observation about this particular loop; but one can say more
Some Things That Do Not Change

• “The concatenation of the strings” does not change
  – *this* \( \times q\) does not change
  – True, and a *stronger* useful observation about this loop because it *implies* the previous observation about the sum of the lengths
    • In other words, if *this* \( \times q\) does not change, then \(|this| + |q|\) also does not change; but not vice versa
How To Express an Invariant

• How do we say “the concatenation of the strings does not change”?  
  – We need to talk about both:
    • The *current values* of the variables
    • The *original values* of the variables, just before the loop condition was *first* tested (variable names prefixed with #)

\[
\textit{this} \; \times \; q = \#\textit{this} \; \times \; \#q
\]
Example #1: Method Body

/**
  * @updates this, q
  * @maintains
  * this * q = #this * #q
  */

while (q.length() > 0) {
  T x = q.dequeue();
  this.enqueue(x);
}
Example #1: Method Body

/**
 * @updates this, q
 * @maintains ...
 * this * q = #this * #q
 */

while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}

This Javadoc tag introduces the list of variables whose values might change in some iteration.
Example #1: Method Body

```java
/**
 * @updates this, q
 * @maintains
 * this * q = this * #q
 */
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
```

Any variable in scope that is not listed as an *updates-mode* variable is, by default, a *restores-mode* variable, meaning the loop body does not change its value.
Example #1: Method Body

```java
/**
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 */
while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
```

This Javadoc tag introduces the claim that the following loop “maintains” the property, i.e., it is a loop invariant.
Example #1: Method Body

/**
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 */

while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
Using a Loop Invariant

• If you have a strong enough loop invariant, you can *trace over* a loop in a single step, and predict the values of the variables when it terminates—without tracing through the loop body even once
Example #1: Method Body

/**
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 */

while (q.length() > 0) {
    ...
}

Pretend you cannot see the loop body. Can you still trace over this loop?
```c
/**
 * @maintains
 * this * q = #this * #q
 */
while (q.length() > 0) {
    ...
}
this = q =
```

When execution reaches this point, we know two things...
/**
 * @maintains
 * this * q = #this * #q
 */
while (q.length() > 0) {
...
}
this = q =

We know (1) the loop invariant is true, so:
this * q = #this * #q
\[
\text{this} = \langle 1, 2, 3 \rangle \\
q = \langle 4, 5, 6 \rangle
\]

```java
/**
 * @maintains
 * this * q = this * #q
 */
while (q.length() > 0) {

... 

}

\text{this} = \q 
```

We also know (2) the loop condition is false, so:

\[ |q| \leq 0 \]
Combining (1) and (2), the only values the variables can possibly have at this point are these.
Justification for (1)

• The loop invariant is true just after the loop terminates—if the code that tests the loop condition does not change the value of any variable appearing in the loop invariant
The Loop Invariant Picture

If the loop invariant is true at the two red points, and "test" updates nothing...

\[ \text{while} \ (\text{test}) \ {\{} \]
\[ \text{loop-body} \]
\[ {\}} \]

...then the loop invariant is true at the two green points.
Justification for (1)

- **Best practice**: Code that tests the loop condition should not update any variables appearing in the loop invariant
  - Easy way to achieve this: the test should not update *any variables at all*
Justification for (2)

• The loop does not terminate until and unless the loop condition is false
  – However, a loop might never terminate; so you need to show that it does
  – This is similar to how you show a recursive method terminates
Loop Termination

• To show that a loop terminates, it is sufficient to provide a progress metric (a.k.a. termination function, a.k.a. variant function)
  – An integer-valued function of the variables in scope (where the loop appears in the code)
  – Always non-negative
  – Always decreases when the loop body is executed once
/**
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 * @decreases
 * |q|
 */

while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
Example #1: Method Body

/ **
 * @updates this, q
 * @maintains
 * this * q = #this * #q
 * @decreases
 * |q|
 */

while (q.length() > 0) {
    T x = q.dequeue();
    this.enqueue(x);
}
Conclusion

• Even if you do not choose to write down a loop invariant or progress metric, if you think about loops in these terms it can help you avoid errors and bad practices in loop code
  – Off-by-one errors
  – Wrong/missing code in the loop body
  – Declarations of variables outside the loop that are only used inside the loop body