Specifications, continued

Announcements
- HW1 was due today at 2
- HW2 will be out tonight
- Grades on Lab1, Lab2 and HW0 coming soon in Homework server
- Quiz 2 today

Outline
- Specifications
  - Benefits of specifications
  - Specification conventions
    - Javadoc
    - JML
    - PoS specifications
  - Specification strength
  - Substitutability
  - Comparing specifications

Specifications
- Modularity
  - A client module can be developed in parallel with a server module
- Abstraction
- Enable reasoning about correctness

Javadoc
- Javadoc convention
  - Method’s type signature
  - Text description of what method does
  - Parameters: text description of what gets passed
  - Return: text description of what gets returned
  - Throws: list of exceptions that may get thrown

Example: Javadoc for String.substring

public String substring(int beginIndex)

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of the string.

Parameters:
- beginIndex --- the beginning index, inclusive.

Returns:
- the specified substring.

Throws:
- IndexOutOfBoundsException --- if beginIndex is negative or larger than the length of this String object.
PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
  - requires: clause spells out constraints on client
- The postcondition
  - modifies: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - throws: lists possible exceptions
  - effects: describes final state of modified objects
  - returns: describes return value

Example 0

String substring(int beginIndex)

requires: none
modifies: none
effects: none
returns: new string with same value as the substring beginning at beginIndex and extending until the end of current string
throws: IndexOutOfBoundsException --- if beginIndex is negative or larger than the length of this String object.

Example 1

static <T> int change(List<T> lst, T old, T newelt)

requires: lst, old, newelt are non-null. old occurs in lst.
modifies: lst
effects: change first occurrence of old in lst with newelt. makes no other changes.
returns: position of element in lst that was old and now newelt

Example 2

static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2)

requires: lst1, lst2 are non-null. lst1 and lst2 are same size.
modifies: none
effects: none
returns: a new list of the same size, whose i-the element is the sum of the i-the elements of lst1 and lst2

Aside: Autoboxing and Unboxing

- Boxed primitives. E.g., int vs. Integer
- Autoboxing: automatic conversion from primitive to boxed type
  - Java generics require reference type arguments
    ArrayList<Integer> al = new ... for (int i=0; i<10; i++) al.add(i);
- Unboxing: automatic conversion from boxed type to primitive
  res.add(lst1.get(i) + lst2.get(i))

Example 3

static void listAdd2(List<Integer> lst1, List<Integer> lst2)

requires: ??
modifies: ??
effects: ??
returns: ??

static void listAdd(List<Integer> lst1, List<Integer> lst2) {
  for (int i = 0; i < lst1.size(); i++) {
    lst1.set(i, lst1.get(i) + lst2.get(i));
  }
}
Example 4

```java
static void uniquefy(List<Integer> lst)
    requires: ?
    modifies: ?
    effects: ?
    returns: ?
{
    for (int i = 0; i < lst.size()-1; i++)
        if (lst.get(i) == lst.get(i+1))
            lst.remove(i);
}
```

Example 5

```java
private static void swap(int[] a, int i, int j)
    requires: ?
    modifies: ?
    effects: ?
    returns: ?
{
    int tmp = a[j];
    a[j] = a[i];
    a[i] = tmp;
}
```

Example 6

```java
private static void selection(int[] a)
    requires: ?
    modifies: ?
    effects: ?
    returns: ?
{
    for (int i = 0; i < a.length; i++)
        int min = i;
        for (int j = i+1; j < a.length; j++)
            if (a[j] < a[min]) min = j;
    swap(a, i, min);
}
```

Javadoc for java.util.binarysearch

```java
public static int binarySearch(int[] a, int key)
    Searches the specified array of ints for the specified value using
    the binary search algorithm. The array must be sorted (as by the
    sort method, above) prior to making this call. If it is not sorted,
    the results are undefined. If the array contains multiple elements
    with the specified value, there is no guarantee which one will be
    found.
    Parameters:
        a - the array to be searched.
        key - the value to be searched for.
    Returns:
        index of the search key, if it is contained in the array;
        otherwise, -(insertion point) - 1). The insertion
        point is defined as the point at which the key would be inserted
        into the array; the index of the first element greater than the
        key, or a.length if all elements in the array are less than the
        specified key. Note that this guarantees that the return value
        will be >= 0 if and only if the key is found.
    So, what is wrong with this spec?
```

Better binarySearch Specification

```java
public static int binarySearch(int[] a, int key)
    Precondition:
        requires: a is sorted in ascending order
    Postcondition:
        modifies: none
        effects: none
        returns: i such that a[i] = key if such an i exists
                 a negative value otherwise
```

Fall 15 CSCI 2600, A Milanova (due to Michael Ernst)
**Shall We Check requires Clause?**

- If client (i.e., caller) fails to provide the preconditions, method can do anything — throw an exception or pass bad value back
- It is polite to check
- Checking preconditions
  - Makes an implementation more robust
  - Provides feedback to the client
  - Avoids silent errors

**Rule**

- If private method, may not check
- If public method, do check, unless such a check is expensive
  - E.g., requires: lst is non-null. Check
  - E.g., requires: lst contains positive integers. Don’t check

**Example 1:** requires: lst, old, newelt are non-null. old occurs in lst. Check?

**Example 2:** requires: lst1, lst2 are non-null. lst1 and lst2 are same size. Check?

**binarySearch:** requires: a is sorted. Check?

**The JML Convention**

- Javadocs and PoS specifications are not “machine-readable”
- JML (Java Modeling Language) is a formal specification language
  - Builds on the ideas of Hoare logic
  - End goal is automatic verification
  - Does implementation obey contract? Given a spec S and implementation I, does I conform to S?
  - Does client obey contract? Does it meet preconditions? Does it expect what method provides?

**The JML Spec for binarySearch**

Precondition:

```
requires: a != null
  && (forall int i; 0 < i && i < a.length;
   a[i-1] <= a[i];
Postcondition:
  ensures: // complex statement…
```

**Dafny Spec for binarySearch**

```
method BinarySearch(a: array<int>, key: int) returns (index: int)
requires a != null && sorted(a);
ensures 0 <= index ==> index < a.Length && a[index] == key;
ensures index < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != key;
```

Difficult problem and an active research area! Look up Sir Tony Hoare’s Grand Challenge
PoS Specifications

- PoS specifications, due to Mike Ernst, are readable and rigorous

- Readability ensures that the spec is a helpful abstraction
- Rigor facilitates reasoning
  - Apply reasoning to prove correctness
  - Compare specifications (more on this later today)

Aside: Specifications and Dynamic Languages

- Specifications are imperative. Even more so for dynamically-typed languages! Why?
  
  ```python
  def add(l1, l2):
      i = 0
      res = []
      for j in l1:
          res.append(j+l2[i]; i=i+1
      return res
  ```

- Start spec with type signature. Then fill in the rest of the clauses. Write spec before code!

Specification for addLists

```java
List<Number> add(List<Number> l1, List<Number> l2)
```

- Requires: ?
- Modifies: ?
- Effects: ?
- Returns: ?

```java
def add(l1, l2):
    i = 0
    res = []
    for j in l1:
        res.append(j+l2[i]; i=i+1
    return res
```

Specification Style

- A method is called for its side effects (effects clause) or its return value (returns clause)
- It is bad style to have both effects and return
- There are exceptions. Can you think of one?
  - E.g., HashMap.put returns the previous value
  - E.g., Box.add returns true if successfully added
- Main point of spec is to be helpful
  - Being overly formal does not help
  - Being too informal does not great either
Sometimes, we need to compare specifications (we’ll see why a little later).

“A is stronger than B” (i.e. A => B) means

- For every implementation I
  - If I satisfies A implies I satisfies B
  - The opposite is not necessarily true

- For every client C
  - “C meets the obligations of B” implies “C meets the obligations of A”
  - The opposite is not necessarily true

### Which One is Stronger?

```
int find(int[] a, int value) {
    for (int i=0; i< a.length; i++) {
        if (a[i] == value) return i;
    }
    return -1;
}
```

**Specification A:**
- requires: a is non-null and value occurs in a
- returns: the smallest index i such that a[i] = value

**Specification B:**
- requires: a is non-null and value occurs in a
- returns: i such that a[i] = value

### Which One is Stronger?

```
String substring(int beginIndex)
```

**Specification A:**
- requires: 0 <= beginIndex <= length of this String object
- returns: new string with same value as the substring beginning at beginIndex and extending until the end of the current string

**Specification B:**
- requires: nothing
- returns: new string with same value as the substring beginning at beginIndex and extending until the end of the current string
- throws: IndexOutOfBoundsException --- if beginIndex is negative or > length of this String object

### Why Care About Specification Strength?

Because of substitutability!

- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - I.e., an implementation that conforms to a stronger specification can be used in a client that expects a weaker specification

---

Fall 15 CSCI 2600, A Milanova (example modified from Michael Ernst)
Substitutability

- Substitutability ensures correct hierarchies
- Client code: `X x; ... x.foo(index);`
  - Client is "polymorphic": written against X, but it is expected to work with any subclass of X
  - A subclass of X, say Y, may have its own implementation of `foo`, `Y.foo(int)`. Client must work correctly with `Y.foo(int)` too!
  - If the spec of `Y.foo(int)` is stronger than the spec of `X.foo(int)` then we can safely substitute `Y.foo(int)` for `X.foo(int)`!

Strengthening and Weakening Specification

- Strengthen a specification
  - Require less of client: fewer/weaker conditions in `requires` clause AND/OR
  - Promise more to client: `throws`, `effects`, `returns`
- Weaken a specification
  - Require more of client: more/stronger conditions to `requires` AND/OR
  - Promise less to client: `throws`, `effects`, `returns` clauses are weaker, easier to write into code

Ease of Use by Client; Ease of Implementation

- A stronger specification is easier to use
  - Client has fewer preconditions to meet
  - Client gets more guarantees in postconditions
  - But a stronger spec is harder to implement!
- Weaker specification is easier to implement
  - Larger set of preconditions, relieves implementation from the burden of handling different cases
  - Easier to guarantee less in postcondition
  - But weaker spec is harder to use

Specification Strength

- Let specification A consist of precondition `P_A` and postcondition `Q_A`
- Let specification B consist of precondition `P_B` and postcondition `Q_B`
- A is stronger than B if (but not only if!)
  - `P_B` is stronger than `P_A` (stronger specifications require less)
  - `Q_A` is stronger than `Q_B` (stronger specifications promise more)

Exercise: Order by Strength

Spec A: requires: a non-negative int argument
  returns: an int in [1..10]
Spec B: requires: int argument
  returns: an int in [2..5]
Spec C: requires: true // the weakest condition
  returns: an int in [2..5]
Spec D: requires: an int in [1..10]
  returns: an int in [1..20]
Group Exercise

Spec A: "returns: an integer ≥ its argument"
Spec B: "returns: a non-negative integer ≥ its argument"
Spec C: "returns: argument + 1"
Spec D: "returns: argument²"
Spec E: "returns: Integer.MAX_VALUE"

Implementations:

<table>
<thead>
<tr>
<th>Code</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code 1: return arg*2;</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code 2: return abs(arg);</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code 3: return arg+5;</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code 4: return arg*arg;</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code 5: return Integer.MAX_VALUE;</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Review

"A is stronger than B" means
- For every implementation I
  - "I satisfies A" implies "I satisfies B"
  - The opposite is not necessarily true!
- A larger world of implementations satisfy the weaker spec B than the stronger spec A
- Consequently, it is easier to implement a weaker spec!
- Weaker specs require more AND/OR
- Weaker specs guarantee less