Recitation 5

Brandon Rozek
rozekb@rpi.edu

Rensselaer Polytechnic Institute, Troy, NY, USA

February 2022

Outline

Three things:

- JavaDoc
- Specifications
- Abstract Data Types

JavaDoc

JavaDoc is a documentation generator whose style has been adopted as the industry standard. The standard tags in the specification with the most common underlined are:

- `@author`: Name of an author. (Multiple tags should be used for multiple authors.)
- `@param`: Name and description of a parameter. (Multiple parameters should appear in the order of the signature of the method)
- `@return`: Return type along with permissible range of values

JavaDoc

The standard tags in the specification with the most common underlined are:

- `@exception`/`@throws`: An exception that may get thrown along with a description of why.
- `@since`: Version of the codebase where this is introduced.
- `@see`: A link pointing to additional documentation.
- `@deprecated`: Mark denoting that the component should no longer be used.

JavaDoc Example

```
/**
** @param degrees An arbitrary double representing an
    angle in degrees.
** @return A double representing the degrees
    normalized to the range [0, 360].
**/
public double normalizeDegrees(double degrees) {
    return degrees - (Math.floor(degrees / 360) * 360);
}
```

Additional Tags for PSoft

Many teams have a style guide or conventions for how they write their specifications. For this class, we will require the following tags:

- `@requires`: The precondition or constraints.
- `@modifies`: List of objects that may be modified by the method.
- `@effects`: Describes the final state (postcondition) of modified objects.

Inspiration coming from Dafny and Hoare logic. Use `none` if a tag above does not apply.

Example

```
/**
** @param c The customer's shopping cart.
** @param i A shopping item.
** @requires i \in Inventory
** @modifies c
** @effects c = \old{c} union {item}
**/
public void addToCart(Cart c, Item i) {
    c.add(i);
}
```

Specifications

Specification Strength

A specification A is *stronger* than B iff:

One of the following is true:

- A has a weaker precondition than B
- A has a stronger postcondition than B

Both of the following are true:

- B does not have a weaker precondition than A
- B does not have a stronger postcondition than A

A stronger specification is more tolerant of inputs and more strict of outputs.

Specification Strength as Logic

Let us denote the precondition as P and the postcondition as Q:

If $((P_B \implies P_A) \wedge (Q_A \implies Q_B))$ *then* A *is stronger than* B.

Specification Strength Example

Spec A:

```
@return y such that y = array[index]
@throws ArrayIndexOutOfBoundsException
    if index < 0 or index >= array.length
```

Spec B:

```
@requires 0 <= index < array.length
@return y such that y = array[index]
```

What can we say about the strength of these specifications?

Specification Strength Practice

Spec C:

```
@requires index >= 0
@return y such that y = array[index]
@throws ArrayIndexOutOfBoundsException
    if index >= array.length
```

Spec D:

```
@return y such that y = null or y = array[index]
```

What can we say about the strength of these specifications?

Type Variances

Lets say we have classes `Student` and `Person` where `Student` is a subtype of ($<:$) `Person`.

Now consider the composite classes `C<Student>` and `C<Person>`:

- The relationship is *covariant* if `C<Student>` $<:$ `C<Person>`
- The relationship is *contravariant* if `C<Person>` $<:$ `C<Student>`
- *Bivariant* is both covariant and contravariant.
- *Invariant* is neither covariant nor contravariant.

Java Arrays are Covariant

In Java, `Student[]` $<:$ `Person[]`.

Any problems with this approach?

```
public class Person {
    class Student extends Person {}

    public static void main(String[] args) {
        Student[] s = new Student[1];
        // Allowed since Student[] <: Person[]
        Person[] p = s;
        p[0] = new Person();
    }
}
```

Generics are Invariant

The following won't compile:

```
public class Person {
  class Student extends Person {}

  public static void main(String[] args) {
    ArrayList<Student> s2 = new ArrayList<Student>();
    ArrayList<Person> p2 = s2;
  }
}
```

Specification and Variance

If specification A is stronger than B then we know:

- Input Contravariance
  - The inputs of A may be a supertype of the inputs of B
  - Weaker precondition, more tolerant inputs.
- Output Covariance
  - The outputs of A may be a subtype of the outputs of B.
  - Stricter postcondition, doesn't violate clients expectations.

Why?

Why do we care about specifications and variances?

Liskov Principle of Substitutability:

> *An object with stronger specification can be substituted for an object with a weaker one without altering correctness.*

Java Modeling Language

A machine-checkable specification language inside Java comments.
Example derived from Wikipedia:

```
public class Banking {
    private /*@ spec_public @*/ int balance;
    //@ public invariant balance >= 0;

    //@ assignable balance;
    //@ ensures balance == 0;
    public Banking() {
        this.balance = 0;
    }

    //@ requires 0 < amount;
    //@ assignable balance;
    //@ ensures balance == \old(balance) + amount;
    public void credit(final int amount) {
        this.balance += amount;
    }
}
```

Dafny Specification

```
method credit(amt: int, old_bal: int) returns
    (new_bal: int)
    requires 0 < amt
    ensures new_bal == old_bal + amt
{
    new_bal := old_bal + amt;
}
```

Abstract Data Types (ADTs)

Control Abstraction

- This is where a method name, signature, and specification is exposed to the client.
- The implementation details is hidden from the user.

Data Abstraction

- This is where the data representation of a class is hidden from the user.
- For example: How are Strings implemented in Java? Fixed array of chars? Linked list?

Abstract Data Types

- An *Abstract Data Type* combines both control abstractions and data abstractions.
- In other words, it encapsulates an object and its operations.

Information hiding is a design principle that segregates and hides the parts of a computer program likely to change.

More Dafny!

- Let's play around with Dafny and see if we can create a banking class.
- We'll likely build on this in later recitations...

Any Questions?