# Brendan Whitaker

## CSE 2221 Homework 13

Professor Bucci

20 March 2017

**1.** We implement the given static method:

```
/**
 * Returns the product of the digits of {@code n}.
 *
 * @param n
 *            {@code NaturalNumber} whose digits to multiply
 * @return the product of the digits of {@code n}
 * @clears n
 * @ensures productOfDigits1 = [product of the digits of n]
 */
private static NaturalNumber productOfDigits1(NaturalNumber n) {
    NaturalNumber product = new NaturalNumber2(1);
    NaturalNumber zero = n.newInstance();
    NaturalNumber rem = n.newInstance();
    if (n.compareTo(zero) > 0) {
        rem.setFromInt(n.divideBy10());
        product.multiply(rem);
        if (n.compareTo(zero) > 0) {
            product.multiply(productOfDigits1(n));
        }
    } else {
        product.copyFrom(zero);
    }
    n.clear();
    return product;
}
```

And part b:

```
    /**
 * Returns the product of the digits of {@code n}.
 *
 * @param n
 *            {@code NaturalNumber} whose digits to multiply
 * @return the product of the digits of {@code n}
 * @ensures productOfDigits2 = [product of the digits of n]
 */
private static NaturalNumber productOfDigits2(NaturalNumber n) {

    NaturalNumber product = new NaturalNumber2(1);
    NaturalNumber zero = n.newInstance();
    NaturalNumber rem = n.newInstance();
    if (n.compareTo(zero) > 0) {
        int remInt = n.divideBy10();
        rem.setFromInt(remInt);
        product.multiply(rem);
        if (n.compareTo(zero) > 0) {
            product.multiply(productOfDigits2(n));
        }
        n.multiplyBy10(remInt);
    } else {
        product.copyFrom(zero);
    }
    return product;
}
```

**2.** We implement the given static method:

```
/**
 * Reports the value of {@code n} as an {@code int},
 * when {@code n} is small
 * enough.
 *
 * @param n
 *            the given {@code NaturalNumber}
 * @return the value
 * @requires n <= Integer.MAX_VALUE
 * @ensures toInt = n
 */
private static int toInt(NaturalNumber n) {
    int returnInt = 0;
    NaturalNumber zero = n.newInstance();
    if (n.compareTo(zero) > 0) {
        int rem = n.divideBy10();
        returnInt = toInt(n);
        returnInt *= 10;
        returnInt += rem;
        n.multiplyBy10(rem);
    }
    return returnInt;
}
```

**3.** We implement the given static method:

```
/**
 * Reports whether the given tag appears in the
 * given {@code XMLTree}.
 *
 * @param xml
 *            the {@code XMLTree}
 * @param tag
 *            the tag name
 * @return true if the given tag appears in the
 * given {@code XMLTree}, false
 *         otherwise
 * @ensures <pre>
 * findTag =
 *    [true if the given tag appears in the given
 * {@code XMLTree}, false otherwise]
 * </pre>
 */
private static boolean findTag(XMLTree xml, String tag) {
    int numChild = xml.numberOfChildren();
    boolean match = false;
    if (xml.label().equals(tag)) {
        match = true;
    } else {
        for (int i = 0; i < numChild; i++) {
            if (xml.child(i).isTag()) {
                match = findTag(xml.child(i), tag);
            }
        }
    }
    return match;
}
```

**4.** We define some important terms:

**Design-By-Contract:** describes the behavior between the client and implement, where each has the responsibility to meet the precondition and postcondition, respectively, and the client and implement can assume the postcondition and precondition is met, respectively, when the method is called/returned
**Precondition:** sets responsibilities of the client calling a method to meet in order for the implementation to return an appropriate result
**Postcondition:** sets responsibilities of the method to return an appropriate result to the client code
**Testing:** technique for trying to disprove that a method body is correct for the method contract; goal is to show that the method body does not correctly implement the contract
**Debugging:** technique for trying to fix and repair bugs and defects in code so that the method correctly returns all appropriate values
**Parameter Mode:** summarize in a method contract a way that a method might change the value of the corresponding argument
**Clears:** resets the parameter to an initial value for its type (i.e. the no-arg constructor value). This is not necessarily the value of the original argument.
**Replaces:** parameter has a value that might be changed from its incoming value, but the methods behavior does not depend on its incoming value.
**Restores:** the method makes sure to restore the parameter to its incoming value (this is the default parameter mode).
**Updates:** parameter has a value that might be changed from its incoming value, and the method's behavior does depend on its incoming value.
**Immutable Type:** types for which no method might change the value of the receiver, or any other argument of that type.
**Primitive Type:** built-in type (boolean, char, int, double).
**Reference Type:** different from primitive as the values are stored in a memory address, reference value is created to reference the memory address where the object value is stored. The value of the reference type is not the object value!
**Object:** the location in memory where a references variable's value is stored.
**Aliasing:** Occurs when a reference type is mutable and two reference types point to the same object. If a method change the object value of one reference variable, it changes for both references.
**Declared Type (static):** the name of the interface or class which is declaring a variable. i.e. "NaturalNumber" in "NaturalNumber k = new NaturalNumber2()".
**Object Type (dynamic):** name of the constructor for which a variable is instantiated, i.e. "NaturalNumber2" in "NaturalNumber k = new NaturalNumber2()".
**Implements:** relation held between a class and an interface where the class contains code for the behavior specified in the interface, can separate contracts from implementations, only instance methods.
**Extends:** relation held between two interfaces or two classes, if one extends

another, the first inherits all the methods of the other.
**Method Overriding:** provides new method bodies for methods which are already implemented in a class which is being extended.
**Subinterface:** refers to the interface that extends another.
**Superinterface:** refers to the interface that is being extended compared to another.
**Polymorphism:** when there are multiple methods with the same name, the method that will be called is the one made from the class from which the constructor is called.
**Recursion:** calling a method in itself to achieve repetition in minimal lines of code.

**6.**

*Proof.* From the graphic on slide 2, we know that NaturalNumberKernel extends Standard, and NaturalNumber extends NaturalNumberKernel. By the definition of extends, we know that NaturalNumberKernel inherits all the methods of Standard. Similarly, we know that NaturalNumber inherits all the methods of NaturalNumberKernel. but since the methods of Standard are also methods of NaturalNumberKernel, we know the methods of Standard are inherited by NaturalNumber. Hence by definition again, NaturalNumber extends Standard. □

**7.**

*Proof.* We know from the given diagram that C3 implements I2 and that C4 extends C3. We want to show that C4 implements I2, i.e. that C4 contains code for the method contracts specified in I2. We know since C3 implements I2 that C3 contains code for the method contracts in I2. Also, since C4 extends C3, by definition, C4 inherits all the method bodies in C3, which means it contains code for the method contracts in I2, hence C4 implements I2 by definition. □

*Proof.* From the diagram, we know that I2 extends I1 which means that I2 has all the method contracts contained in I1. We also know that C3 implements I2, which means that C3 contains code for each method contract in I2. But since every method contract in I1 is contained in I2, we know that C3 contains code for each method contract in I1, hence by definition C3 implements I1. □