```
/**
     * 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 ans = new NaturalNumber2(1);
        while (!n.isZero()) {
            NaturalNumber digit = new NaturalNumber2(n.divideBy10());
            ans.multiply(digit);
        }
        return ans;
    }

    /**
     * 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) {

        int digit = n.divideBy10();
        NaturalNumber product = new NaturalNumber2(digit);
        if (!n.isZero()) {
            product.multiply(productOfDigits2(n));
        }
        n.multiplyBy10(digit);
        return product;
    }

    /**
     * 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 lastDigit = n.divideBy10();
        int other = 0;
        if (!n.isZero()) {
            other = toInt(n) * NaturalNumberKernel.RADIX;
        }
        n.multiplyBy10(lastDigit);
        return other + lastDigit;
    }

    /**
     * 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) {

        boolean found = false;
        if (xml.isTag()) {
            if (xml.label().equals(tag)) {
                found = true;
            }
            int i = 0;
            while (!found && i < xml.numberOfChildren()) {
                found = findTag(xml.child(i), tag);
                i++;
            }
        }
        return found;
    }
```

Design-By-Contract is a way of designing software with preconditions and postconditions that the client and implementer can use and are bound by so that assertional thinking is possible.

A precondition is a requirement of what must be true at a certain point in the code for the postcondition to hold true also

A postcondition is a requirement of what must be true after a certain point in the code. The postcondition **must** be true as long as the precondition was true heading into the contract

Testing is the process of ensuring that the software is doing what its supposed to be doing

Debugging is the process of identifying, finding, and correcting a problem in the code

Parameter mode is a way of specifying what happens to a parameter in a method, and how it will look afterwards.

Clears mode is when the parameter is changed to its default value

Restores mode means the parameter in the end state holds the same value as its beginning state: (p = #p)

Replaces mode means the initial value of the parameter does not matter, and the value will be completely replaced by another value.

Updates mode means the value of the parameter is changed based off of the old value: (p = #p + 1)

An immutable type is a reference type whose object value cannot be changed

A primitive type is a type that exists by default in java

A mutable type is a reference type whose object value can be changed

An object is an instance of a class that is stored in fields and has a certain behavior

Aliasing is when two variables have the same reference value

Declared type/static type is the type of an object declared at compile time, and points to a class.

An object type/dynamic type is the type of an object determined at run-time and points to the specific instance of a class.

An implements relationship is how a class provides all the code for a interface

An extends relationship is how an interface can add to another interface or a class can add to another class, inheriting all methods in the parent interface or class.

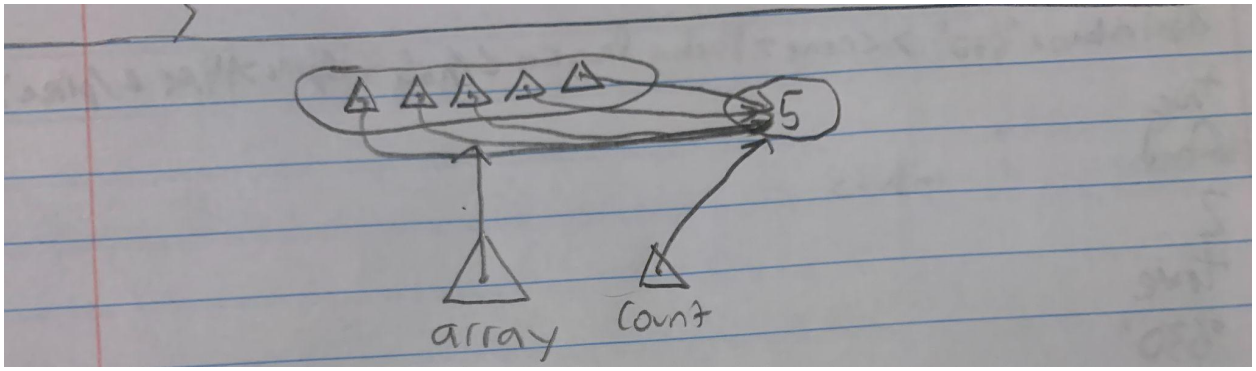
Method overriding is when a subclass has the same method as its superclass, but its body of code carries out the implementation of that method in a different way.

A subinterface/derived interface/child interface is an interface that extends another interface

A superclass/base class/parent class is a class that is extended by another class

Polymorphism happens when an object's static type and dynamic type differ

Recursion is a technique used when a method's definition includes a call to itself



This code is wrong because the array is initialized with references values all aliased to the variable 'count', which is incremented, thus changing the object values of every element in 'array'.
I would change this code by changing line 7 to
`array[i] = new NaturalNumber2(count);`

All of Standard's methods can be used in NaturalNumber, yet NaturalNumber does not explicitly define any of the methods in the Standard. Thus, NaturalNumber must extend Standard in order to use Standard's methods

All of the contracts defined in I2 must be carried out by C3, since C3 implements I2. Because C4 extends C3, every contract fulfilled by C3 is carried over to C4. Thus, C4 implements I2. C3 also implements I1 because every contract in I1 carries over to I2, and if C3 must have code for every contract in I2, then it must have code for every contract in I1.