Chapter 3

Correctness

One of the most appealing aspects of computer programs is that they have a close connection to mathematics, in particular, logic. We use these connections implicitly every day when designing our programs. However, it is worthwhile to deeply understand them so that we can use them to formally reason about our programs and their behavior.

When applying formal reasoning to computer programs, we are interested in determining if a *property* holds of a particular program. There are many sorts of properties about programs that we care to analyze; the two we'll focus on in this course are:

- *Correctness*: does the program produce the desired output?
- *Complexity*: how many resources does the program consume?

Correctness should be important to everyone that writes programs as we care deeply that our program accomplishes the task we set out to do! Previously, you checked the correctness of your program by *testing* it, *i.e.*, running the program on particular inputs and ensured that it produced the output that you expected. Testing is a powerful methodology for ensuring program correctness but it has its downsides. In particular:

- In certain situations, *e.g.*, programs that control rocket ships or automated cars, the cost of testing is prohibitive either in terms of money (the rocket can blow up) or risk (the car could hit someone).
- When developing a large-scale application, we may develop a small piece of it and want to have some assurances that we have done the right thing, but the application is not close to a state in which we can run.
- We would like guidance on how to write our programs beyond simply knowing correlations between inputs and outputs.

As a complement to testing, we can use *mental models of computation* to reason about the behavior of our programs for all possible inputs and without running them explicitly.

3.1 Mental Models of Computation

One view of mathematics is that it is the study of abstractions for the purpose of modeling real-world phenomena. With these models, we can compare, categorize, and predict the behavior of these phenomenon. However, because these models are formed through observations of the phenomenon, there is the potential

for disconnects between the models and reality, *e.g.*, in physics, classical mechanics breaks down in the presence of objects approaching the speed of light. In contrast, the behavior of programming languages are usually designed with a mathematical model of computation in mind. Thus there is no disconnect between how a programming language behaves and our own mental model operates, as long as we have the correct model in mind!

A Note on Paper To practice the skill of programming, we write software artifacts, and we stress that the process of developing these artifacts is invaluable in your development as a computer scientist. Likewise, to practice mathematics, we write stuff down on paper, whether that is diagrams, mathematical definitions, or proofs. Treat any mathematical paper exercise like programming by actually doing said exercise in full detail: pull out paper and write stuff down! It turns out that resorting to paper is also a great way to work out a programming problem, so it's a good habit to get into now if you are not doing it already!

3.1.1 The Substitutive Model of Computation

Recall that in most programming languages, there are two kinds of program fragments:

1. Expressions which evaluate to a value.
2. Statements that do not evaluate to a value but do some meaningful work.

Let's focus on building a model for this first class of program fragments. You are already familiar with this model of computation: it's what you do when you evaluate mathematical expressions. For example, evaluation of the expression `12 + 6 * (4 - 2)` proceeds as follows:

```
    12 + 6 * (4 - 2)
--> 12 + 6 *    2
--> 12 +    12
-->     24
```

Each line represents one step of evaluation we take to evaluate the expression. The final result, `24`, is a value: an expression for which we can no longer take any evaluation steps.

To find and carry out the next step of evaluation, we do the following:

1. Find the next sub-expression to evaluate based off of precedence rules.
2. Carry out the evaluation of that sub-expression to a value.
3. Substitute the value for that sub-expression.

Finally, we repeat this process until we arrive at a final value.

With programming language expressions, the process is identical except that we extend this sort of reasoning to other sorts of expressions, not just arithmetic ones. For example, using the rules of precedence, we know that when evaluating the expression `arr.length > i - 1`, we know that `-` has higher precedence than `>`, so we evaluate `i - 1` first rather than `arr.length > i`. If we did evaluate `arr.length > i`, we would encounter a type error because the result of `arr.lenght > i` is a boolean, but then we try to subtract one from it.

Functions only add a slight complication to this model. For now, we'll only consider pure functions, *i.e.*, functions that only return a value and do not produce any side-effects like mutating (re-assigning) global variables or printing to the console. An example of a pure function is this simple increment function:

```
public static int inc(int x) {
    return x + 1;
}
```

When evaluating an expression containing a function call, we follow the steps above as normal. However, when we evaluate a function, we do the following:

1. Evaluate the arguments to the function call down to values.
2. Substitute the actual values passed to the function for the formal parameters (*i.e.*, variables) of the function.
3. Evaluate the body of the function to a final value.
4. Substitute this resulting value for the function call.

As an second example, consider evaluating the expression, `inc(1) + inc(5*2)`:

```
    inc(1)  + inc(5*2)
--> (1 + 1) + inc(5*2)
-->    2    + inc(5*2)
-->    2    + inc(10)
-->    2    + (10 + 1)
-->    2    +    11
-->         13
```

3.1.2 The Stack and Heap Model of Computation

The substitutive model of computation is a simple model to work with. And indeed, we can use it to reason about our programs when there are no side-effects (which is one of the reasons why we try to minimize the number of variables and assignments in our code). However, we inevitably need to deal with side-effects whether it is dealing with variable that change values, printing text to the console, or reading input from the user. The substitutive model does not deal with side-effects as it deals exclusively with expressions instead of statements whose purpose is to produce side-effects.

We therefore introduce another model of computation, the stack and heap model, which will allow us to reason about side-effects, in particular, variables whose values change over time. Recall that *the stack* contains *activation records* of each currently active function call in the program. These records contain the parameters and local variables for that function. A stack is an appropriate data structure for capturing function calls because function calls obey a first-in-first-out (FIFO) discipline:

- The most recently called function is always the currently active function call.
- When we return from a function, we always return from the most recently called function first.

The heap on the other hand is where all of our dynamically allocated memory resides. In C, we dynamically allocated memory when ever we called `malloc`. In Java, all non-primitive data is allocated on the heap—the most critical difference between Java and C is that you don't have control of where your data resides in memory!

How does the stack and the heap model work? Let's demonstrate it with a step-by-step example. Consider the following code snippet:

```
public class Program {
    public int inc(int x) {
        int y = x + 1;
        return y;
    }
    public static void main(String[] args) {
        int x = 0;
        System.out.println(x);
        System.out.println(inc(x));
        x = inc(x+1);
        System.out.println(x);
        System.out.println(inc(x));
    }
}
```

First, read the code and try to guess what the output of the program will be. Now, we'll verify your hypothesis by tracing through the execution of the program using the stack and heap model of evaluation. We first start by calling the `main` method which takes as a parameter an array of strings collected from the command-line invocation of the program. Our stack and heap at the top of start look as follows:

```
    Stack         Heap
    -----         ----
  --main
  args [ o-]----->[]
```

The stack contains a single frame recording that we are in `main`. The one parameter to `main`, `args`, is a reference to an empty array.

We then begin executing the statements of `main` line by line. The next line is a variable declaration. We add this entry to the stack:

```
    Stack         Heap
    -----         ----
  --main
  args [ o-]----->[]
     x [ 0 ]
```

Because `x` has a primitive type, `int`, the variable on the stack holds the integer value directly rather than holding a reference to a value stored on the heap. On the next line, we print out the current value of `x` which is `0`:

```
    Stack         Heap        Output
    -----         ----        ------
  --main                      0
  args [ o-]----->[]
     x [ 0 ]
```

Next, we call `inc` passing in the current value of `x` which is `0`. Calling `inc` creates a new frame on the stack which contains the lone parameter for `x`. This parameter is loaded with the value of `x`. Note that the `x`s refer to distinct memory locations here.

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]
   x [ 0 ]
--inc
   x [ 0 ]
```

Inside the function, we declare a local variable `y`, adding it to `inc`'s stack frame. `y` is initialized to be one greater than `x`. Again, it is the local `x` value in `inc` that we use rather than `main`'s `x` because `inc` is the currently active function.

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]
   x [ 0 ]
--inc
   y [ 1 ]
```

The function ends by returning the current value of `y` which is `1`. When we return, the stack frame for `inc` is deleted, leaving the stack frame for `main`, and we continue execution of the program where we left off in `main`. Returning from a function works just like the substitutive model: we substitute the result of the function call for the function call itself and continue evaluating the expression. This means that we print out `1` next.

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 0 ]
```

Next, we re-assign the value of `x`. To do this, we call `inc` again passing in `x + 1`. Note that the value of `x` in `main` did not change as a result of the previous function call! This is because `inc` only changes the value of its local variables; it is unable to affect the local variable of `main` directly.

Therefore, when we call `inc` a second time, we pass `1`. Walking through the execution of the function, we return from `inc` in the following state:

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 0 ]
--inc
   x [ 1 ]
   y [ 2 ]
```

From this, we see that we return `2` from the call to `inc`. This value is then stored in `x` (in `main`) as the result of the assignment:

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 2 ]
```

We then we print out the current value of `x` which is 2.

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 2 ]                   2
```

And finally, we print out the result of calling `inc` on `x` one last time. The call from `inc` returns with the following state:

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 2 ]                   2
--inc                        3
   x [ 2 ]
   y [ 3 ]
```

and so we return 3 from `inc` and immediately print that result. Thus, the final state of the stack and heap as well as our output as we leave `main` is:

```
   Stack         Heap        Output
   -----         ----        ------
--main                       0
args [ o-]----->[]           1
   x [ 2 ]                   2
                             3
```

This seems like a lot of work to trace through such simple code, but all these details are necessary to ensure that we arrive at the correct result. This sort of attention to detail is critical for trying to reason about more complicated code where lots of variables and complex data structures are at play.

3.1.3 Objects and the Heap

Recall that values of strings and arrays are allocated on the heap rather than on the stack. Variables of these types are actually references to to these heap-allocated values. For example, after executing the following code snippet:

```
String s = "hello world!";
int[] arr = new int[] { 2, 4, 6 };
```

The state of the stack and heap are:

```
  Stack            Heap
  -----            ----
 s   [ ]-----> "Hello world!"
 arr [ ]-----> { 2, 4, 6 }
```

This logic applies to all objects. All objects are allocated on the heap and variables of these types are actually references to these values.

For example, consider a `Counter` class:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value += 1;
    }
}
```

Let's step through some code using `Counter` and explore how the stack and heap change over time. First, let's initialize a counter:

```
Counter c1 = new Counter();
```

We create a local variable for `c1` on the stack. Its eventual value is the result of the right-hand side of the assignment which is a **`new`** expression. When evaluating a 'new' expression, the following happens:

1. We evaluate the arguments of the **`new`** expression down to a value.
2. We allocate space on the heap for a `Counter`.
3. We then invoke the constructor of the `Counter` like a method or function.
4. We return a reference to the newly allocated and initialized `Counter` when exiting the constructor.

After the second step of this process, we enter the 'Counter' constructor with the following stack and heap:

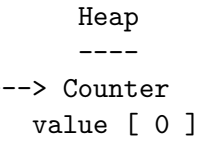
```
  Stack               Heap
  -----               ----
 c1   [ ? ]  |-----> Counter
 -- Counter  |      value [ 0 ]
 this [   ]--|
```

We allocate a stack frame for the constructor call. The constructor has no arguments, but recall that we use **`this`** to refer to the newly created `Counter` within the constructor. In memory, we realize this by treating

`this` as a variable—think of it as a hidden, first parameter to the constructor. This variable is loaded with a reference to the newly allocated `Counter` on the heap. Note that the heap allocation is a chunk of memory containing not only a field for the `value` field of the `Counter` but also a tag stating that the chunk of memory is indeed a `Counter`. Therefore, each object has a notion of its own type at runtime which we can query using the `instanceof` operator:

```
System.out.println(c1 instanceof Counter);    // prints "true"
```

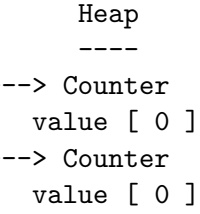
After returning from the constructor, the stack and heap look like this:



Let's create another `Counter`:

```
Counter c2 = new Counter();
```

After executing the constructor, we arrive at the following stack and heap:

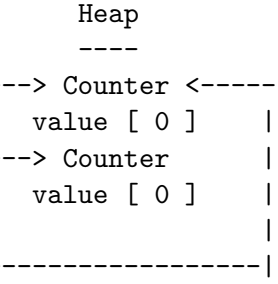


Note that `c1` and `c2` each have their own distinct `value` field, reinforcing the idea that each object has their own copy of the fields specified by their associated class.

Next, let's increment `c1`:

```
c1.increment();
```

Method invocations operate similarly to functions: we evaluate arguments, create a stack frame, and copy over arguments. The stack and heap look like this after entering the `increment` method:



While `increment` takes no parameters, the `this` variable is available in `increment` just like how it is available in the constructor. We load the `this` variable with (a reference to) the `Counter` object that we called `increment` on; in this case it is `c1`. Therefore, the effect of the single statement in `Counter` is to mutate `c1`'s `value` field. After returning from `increment`, we arrive at this stack and heap state:

```
  Stack                Heap
  -----                ----
 c1    [  ]---------> Counter
                     value [ 1 ]
 c2    [  ]---------> Counter
                     value [ 0 ]
```

Note again that `value` fields of `c1` are distinct: we only incremented `c1` with that call to `increment` and did not modify 'c2'.

Exercise (Stack-and-Heap Tracing) Consider the following code that utilizes the `Counter` class:

```
public class Program {

    public void doWork(Counter c) {
        c.increment();
        c = new Counter();
        // Point A
        c.increment();
        c.increment();
    }

    public static void main(String[] args) {
        Counter c = new Counter();
        doWork(c);
        // Point B
    }
}
```

1. Give a stack-and-heap diagram describing the state of memory at `/* Point A */` in the code above.
2. What is the value of `c`'s `value` field at `/* Point B */`?

3.2 Pre-Conditions, Post-Conditions, and Invariants

With appropriate mental models of computation, we can now reason about the behavior of our code. To do so, we specify *properties* about how our programs should behave and then *prove* that our programs obey those properties. Because our programs are complex, it is usually infeasible to directly prove their correctness. Instead, we usually define and prove several properties that, when taken together, imply the correctness of our program, or at the very least, give us high confidence that our program is performing as desired.

Reasoning about program correctness is particularly important for a pair of reasons

- **Ensuring quality code**. Bugs cost money, and bugs cost more money the later they are found in the development process. Finding a logic bug in our reasoning during program development is less costly than discovering it once the product has been deployed.

- **Driving program design**. It is instrumental in program design to have a crisp idea of how your program ought to behave. This knowledge helps concretely when actually designing and writing the program. As a simple example, suppose that you are writing a function that sums up the elements of the an array. You might have the following set up:

  ```
  int sum = 0;
  for (int i = 0; i < arr.length; i++) {
      // ??
  }
  ```

  If you recognize the *loop invariant* that `sum` always contains the sum of the numbers we've seen so far", then writing the loop in such a way to maintain the invariant is easy:

  ```
  int sum = 0;
  for (int i = 0; i < arr.length; i++) {
      sum += arr[i];
  }
  ```

  Furthermore, you know your loop does the right thing because your loop invariant that you've ensured guarantees that `sum` contains the desired value.

3.2.1 Pre-Conditions and Post-Conditions

We can view the signature of a function or a method as a *contract* between the user of the function, the *caller*, and the and implementor of the function, the *callee*. In this view, a *pre-condition* of a function is an obligation that the caller must fulfill. In contrast, a *post-condition* of a function is an obligation that the callee must fulfill provided that the caller fulfills the pre-conditions of the function. For example, consider a function that computes $n!$ called `factorial`:

```
/**
 * @param n the input where n >= 0
 * @return n!
 */
public static long factorial(int n) { /* ... */ }
```

The pre-conditions to `factorial` are:

- The caller provides a single argument.
- That argument is an integer.
- That argument must be non-negative.

If the user meets these requirements, then the post-condition to `factorial` is the function produces $n!$ as expected. Note that some of the pre-conditions and post-conditions are enforced by Java's type system, *e.g.*, the function takes one argument that is an `int` and returns a `long`. Others are not enforced by the type system and are documented explicitly, *e.g.*, the argument must be non-negative and the function produces `n!`. Still others are not even documented as they are understood by convention. For example, because `factorial` returns a `long`, it will only work as long as $n \leq 49$ (because $50! > 2^{63} - 1$, the maximum value a `long` can hold).

From the caller's perspective, pre-conditions and post-conditions inform the caller how to use a function and what to expect. From the callee's perspective, pre-conditions and post-conditions inform the callee what cases their function must be able to handle. In particular, by specifying a series of pre-conditions, the callee restricts these cases, making their job of implementation more managable. In the extreme case, pre-conditions keep the callee from having to "make up" outputs in the presence of invalid inputs such as negative values of *n* for `factorial`.

3.2.2 Small-Scale Reasoning

Pre-conditions and post-conditions allow us to reason about the correctness of *using* a method. However, how do we reason about correctness *within* a method? To do so, we must apply similar sorts of reasoning to program constructs other than method invocations.

Initial Values Let's start with the top of a method declaration, *e.g.*:

```
public void foo(int x, String s) {
    // ???
}
```

What can we say about the possible values of `x` and `s` as we enter the method? In the absence of any pre-conditions, the variables can take on any valid value that their types allow. This means that `x` can be any integer (recall that integers in Java are signed, 32-bit values) and `s` can be any string or `null` (as strings are a reference type). We frequently will want to avoid having to consider that an object may be `null` by assigning the function an appropriate pre-condition, for example:

```
/**
 * ...
 * @param s a non-null input string
 */
public void foo(int x, String s) {
    // ???
}
```

Assignments, Initialization, and Sequencing Consider the variable initialization/assignment statement:

```
int y = 5;
```

What do we learn about our program *after execution* of the initialization? Variable initialization introduces a new variable along with a value for that variable. So after the initialization, we know that `y` is `5`. However, what if we do the following assignment instead?

```
int y = x;
```

From above, we know that `x` is any integer. Therefore, after the assignment, `y` is equal to `x`, also any integer. Whatever value `y` had before is now gone; `y` now has the value `5`.

This seems like a trivial discovery, but this sort of reasoning gets more complicated as we chain together statements. For example, what about the following chain of assignments?

```
int y = 5;
x = x + y;
```

On the first line, we establish that y is 5. On the second line, we add y to x, so now what can we say about x? We can say several things:

- Coarsely speaking, x is still any integer value.
- More specifically, x is 5 greater than it originally was.
- In terms of ranges, x is any integer value in the range `Integer.MIN_VALUE + 5` to `Integer.MAX_VALUE` assuming that we do not overflow x.

Note that to arrive at these conclusions, we *compose* our reasoning between the two statements. We analyze the first statement and then take what we learn from that statement (y is 5) and use it to analyze the second statement (x is incremented by 5). This composition of reasoning between statements is the cornerstone of reasoning about imperative programs.

Conditional Statements Consider a basic conditional statement:

```
if (x < 5) {
    // ??
} else {
    // ??
}
```

What can we say about x during execution of the conditional? If we happen to go into the if-branch, we know because of how the conditional operates, that the boolean expression must have evaluated to `true`. This means that inside the if-branch, x is certainly less than 5. Likewise, if we go into the else-branch, we know that the boolean expression must have evaluated to `false`. This means that inside the else-branch, x is certainly equal to or greater than5. Thus we can note that:

```
if (x < 5) {
    // Here, x < 5
    // ...
} else {
    // Here, x >= 5
    // ...
}
```

Regardless of the branch we go into, we know that control flows to the line after the if-statement. Therefore, the if-statement can introduce some uncertainty into what we know about our variables. Let's now make our conditional slightly more interesting.

```
if (x < 5) {
    y = 0;
} else {
    y = x + 1;
}
```

Now what can we say? If `x < 5`, then `y` is set to `0`. Otherwise, if `x >= 5`, then oy is set to `1` greater than `x`. Combining both pieces of information, we can conclude that after the if-statement that `y` is either `0` or it is at least `6`.

Generalizing, we note that `x < 5` could be any boolean expression `b`. Therefore, we can reason about the conditional as follows:

```
if (b) {
    // we know b is true...
} else {
    // we know b is false...
}
// we know either the if-branch or the else-branch happened...
```

Importantly, after the conditional, we know that we entered one of the two branches. Therefore, after the conditional, we need to consider both situations in which we entered either the if-branch or the else-branch.

We can generalize this behavior to the alternative forms of conditional statements that Java offers. For example with multiple else-if branches, we can reason as follows:

```
if (b1) {
    // we know b1 is true...
} else if (b2) {
    // we know b1 is false, b2 is true...
} else if (b3) {
    // we know b1, b2 is false, b3 is true...
} else {
    // we know b1, b2, b3 is false...
}
```

Note how this contrasts with the absence of else-branches:

```
if (b1) {
    // we know b1 is true...
}
if (b2) {
    // we know b1 is true, nothing about b1...
}
if (b3) {
    // we know b3 is true, nothing about b1, b2...
}
```

As a concrete example, we'll commonly want to establish that a reference type is non-`null` before using it. The following snippet of code guarantees this:

```
if (s == null) {
    s = "default";
}
```

In the absence of a pre-condition or some other knowledge about `s`, it could have any string value, including `null`. If `s` is `null`, we enter the if-branch and assign it a non-`null` value. Otherwise, `s` was already non-null by definition and we skip the conditional altogether.

Loops How do we reason about loops? Unlike conditionals, they do not have "straight-line" behavior—by definition, they loop! What can we say about a program after a loop terminates?

We establish such properties as *loop invariants*. Loop invariants are properties that hold:

- *Before* execution of the loop.
- *After each iteration* of the loop.
- *After* execution of the loop.

Note that the first two conditions implies the third. If a property holds before we enter a loop and then holds for each iteration of the loop, then it must hold after the loop finishes—the proof follows by *induction* on the number of iterations of the loop.

As an example, consider our motivating example from the beginning of this chapter. Suppose that we have the code:

```
int[] arr = /* ... */ ;
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
```

Let our loop invariant be that "`sum` contains the sum of the numbers of `arr` that we have visited so far". Our loop invariant holds initially—we haven't seen any numbers yet so the sum is zero. On every iteration of the loop, we add the current number in the array to the sum. This maintains the invariant that the sum is indeed the sum of the numbers we have seen so far. Therefore, we know that the invariant is preserved and sum indeed contains the sum of the numbers found in the array.

Class Invariants Loop invariants are properties that are always true of a loop. We verify that they hold by ensuring that the property holds initially and then on every iteration of the loop. We can extend similar sorts of reasoning to verify the class invariants that we discussed when we talked about classes earlier. A class invariant is like a loop invariant except that instead of being true of a loop, it is true of any instance of a class.

As an example of a class invariant, suppose that we writing a weight-tracking application and want to represent a person in this application as a class, `Person`. We might begin the definition of the class as follows:

```
public class Person {
    private int weight;
    // ...
}
```

One immediate class invariant on `Person` is that `weight` should never go negative, that is `weight >= 0`.

Like loop invariants, we verify that the class invariant holds *initially* upon creation of the object. We then verify that every method *preserves* the class invariant thereby ensuring that the invariant always holds. Note that our particular invariant relies critically on `weight` being `private` so that arbitrary clients cannot simply mutate `weight` to be negative. For example, the constructor of the `Person` might look like:

```
public Person(int weight) {
    this.weight = weight;
}
```

The `weight` field is initialized by whomever calls the constructor. However, if they pass in a negative weight, this will violate our class invariant. To prevent this, we can guard against a negative weight and either (1) set the weight to `0` or (2) *throw an exception*.

```
public Person(int weight) {
    if (weight < 0) {
        throw new IllegalArgumentException();
    }
}
```

An exception signals an *exceptional case* in the execution of our code that we may (or may not) be able to recover from. In this case, the user of the `Person` class has violated our class invariant by giving an inappropriate weight. We signal an error by *throwing* an exception object. The syntax of a `throws` statement is:

```
throw <expr>;
```

Where `<expr>` is an expression that evaluates to an `Exception` object. There are a number of such objects available in the standard library to represent a wide variety of common scenarios. `IllegalArgumentException` represents the situation where a caller of a method or constructor has passed along illegal values. Note that here we are constructing a new `IllegalArgumentException` object with a parameterless constructor. The `IllegalArgumentException` class also specifies a one-argument constructor that is a string that documents more precisely the error that occurred, *e.g.*,

```
throw new IllegalArgumentException("Negative weight given");
```

With this guard, we ensure that if we successfully create a `Person` object, then we know the class invariant holds.

Now we just need to verify that each method of the `Person` class preserves this class invariant. For example, consider a method to update the person's weight:

```
public void updateWeight(int delta) {
    weight -= delta;
    if (weight < 0) { weight = 0; }
}
```

`updateWeight` simply subtracts its argument from `weight`. However, our class invariant becomes broken if `weight - delta` is negative. The method fixes this up by setting the `weight` to zero in this case. Note that a class invariant may be broken *in the middle of a method*. However, the class invariant must hold true by the time the method exits.

3.3 Invariants and Program Design

In the previous lab, we reasoned about programs using *pre-conditions*, *post-conditions*, and *invariants*. We did this in order to convince ourselves that our programs were correct; usually these conditions implied the overall correctness property of our program that we wanted to enforce. However, these reasoning tools are not just useful for ensuring that a created program is correct after-the-fact. We can use them, in particular, loop invariants to decompose problems and write solutions to them that are correct by construction.

3.3.1 Iterative Design with Loop Invariants

As discussed previously, loop invariants are properties that obey three rules:

1. A loop invariant is true *entering* the loop.
2. A loop invariant is true *at the end of every iteration* of the loop.
3. A loop invariant is true *after execution* of the loop.

Where the first two rules imply the third. If we identify that a solution requires a repetitive action to be performed, we can summarize the results of this repetitive action into a *loop invariant* that our loop will need to fulfill. By doing this, designing the loop becomes significantly easier.

As an example, consider the simple problem of writing a non-recursive version of the function `int factorial(int n)` that returns $n!$. For now we'll assume as a pre-condition that the user only passes `n >= 0` so that we do not need to worry about error-handling. This function clearly requires repetitive behavior, multiplying successive numbers from `1` to `n`, insinuating that a for-loop is a good construct to use.

```
int factorial(int n) {
    // ...
    for (int i = 1; i <= n; i++) {
        // ...
    }
    // ...
}
```

However, how do we fill in this skeleton? First we note that we need to perform some calculation on each number "visited" in the for-loop. This requires an additional local variable that we'll call `ret`. We initialize `ret` before the loop and return it afterwards.

```
int factorial(int n) {
    int ret = // ...
    for (int i = 1; i <= n; i++) {
        // ...
    }
    return ret;
}
```

Now, what loop invariant shall we enforce of our program? It needs to deal with the contents of `ret` and the behavior of the program we want to write. If our program needs to calculate `n!` and proceeds through the numbers `1` through `n` to do so, the following is a reasonable loop invariant to enforce:

> `ret` contains the product of the numbers from `1` to `k` where `k` is the count of numbers we have seen so far.

In light of this, we can apply our three rules of loop invariants to design our program:

1. **Entering the loop**. When we first enter the loop, we haven't seen any numbers yet. Therefore, we need to choose an initial value for `ret` that reflects this. We know that `0! = 1`, therefore we choose `ret` to be `0`.

2. **At the end of iteration of the loop**. On the `i`-th iteration of the loop, we see number `i`. How do we update `ret` so that our loop invariant holds? We merely multiply `ret` by `i` and store the result back in `ret`!

3. **After execution** By satisfying the two previous rules—preserving the invariant while entering the loop and on every iteration—we know that the loop invariant is preserved when the loop exits. Interpreting the loop invariant, we see that this implies that `ret` contains `n!`!

This *invariant-driven design* results in the following program:

```
int factorial(int n) {
    int ret = 1
    for (int i = 1; i <= n; i++) {
        ret *= i;
    }
    return ret;
}
```

Of course, you could have written this loop without explicitly going through process. But by following this invariant-driven design process mechanically, we can consistently produce loops that are correct, especially when the programs themselves get more complicated!

Exercise (Loop Invariant Verification) Consider the following implementation of the `max` function:

```
public static int max(int[] arr) {
    int ret = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < ret) {
            ret = arr[i];
        }
    }
    return ret;
}
```

Give an appropriate loop invariant for `max` in terms of `ret` that implies the correctness of the `max` function if it holds. Does the `max` function enforce this loop invariant properly?