This Scala notebook uses *BeakerX*, a Two Sigma Open Source project that enhances Jupyter.

http://beakerx.com/ (http://beakerx.com/)

```
In [1]: scala.util.Properties.versionMsg
```

```
Out[1]: Scala library version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL
```

The Design Recipe

Step 1: Analyze the Problem

- What are the objects in the problem domain?
- What data types we will use to represent them?

Step 2: Create a Contract

- What are the names of our functions and their parameters?
- What are the requirements of the data they consume and produce?
- What is the meaning of what our program computes?

Step 3+: Iterate until Correct

Repeat the following steps until we are confident in our program's correctness:

1. Write some *tests* (start with example inputs/outputs)
2. Sketch a function *template*
3. *Define* the function

Example: Profit of a Movie Theater

The owner of a movie theater collected the following data:

- At $5.00 per ticket, 120 people attend a performance.
- Decreasing by $0.10 increases attendance by 15 people.
- A performance costs $180 plus $0.04 per attendee.
- Define a function to calculate the exact relationship between ticket price and profit.

Problem Statement taken from *How to Design Programs* (2001)

Analysis

- What are the objects in the problem domain?
- What data types we will use to represent them?

What types of variables are we working with?

- We want to compute `profit`
- `profit` is defined as `revenue - cost`
- `cost` is dependent on the `attendendance`

So we are working with *monetary values* and *counts of attendees*.

Attendees

- Counts of people are *whole numbers*.
- Assume theaters seat less than a thousand people.
- We can represent this count with an `Int`.

Monetary values

- U.S. dollar values can be fractional (e.g., $0.10).
- Can't charge monetary fractions smaller than $0.01.
- Assume monetary values are less than $10 million per performance.
- We can represent a monetary value in *cents* as an `Int`.

Function Contract

- What is the name of the function?
  - What considerations should go into the names we choose?
- What are the static types of the arguments that our function consumes?
  - What other constraints must hold on the values it consumes?
- What is the static type of its result?
  - What else does it ensure about its result?
- What is the purpose of our function?

What is a reasonable contract of the `attendance` function?

In [2]:
```
def attendance(ticketPrice: Int): Int = ???
```

Out[2]:
```
attendance: (ticketPrice: Int)Int
```

Using Scala's ??? Placeholder

`???` is a built-in placeholder symbol in Scala. `??? : Int` is a placeholder for an expression that will have the static type of `Int`. Using the `???` placeholder allows us to write Scala code that compiles even if it's not yet fully implemented. Evaluating `???` (either directly or indirectly) will result in a run-time error.

In [3]:
```
???  // directly evaluating the placeholder value
```

```
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  ... 46 elided
```

In [4]:
```
attendance(100)  // indirectly evaluating the placeholder value
```

```
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at attendance(<console>:88)
  ... 46 elided
```

Is the function signature of `attendence` a sufficient contract?

Preconditions

We express assertions about the input to a function, or its *preconditions*, using Scala's `require` construct.

What are the *preconditions* of `attendance`?

Postconditions

We express assertions about the result of a function, or its *postconditions*, using Scala's `ensuring` construct.

What are the *postconditions* of `attendance`?

In [5]:
```
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  ??? : Int
} ensuring (result => result >= 0)
```

Out[5]:
```
attendance: (ticketPrice: Int)Int
```

Statement of Purpose

In [6]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  ??? : Int
} ensuring (result => result >= 0)
```

Out[6]:
```
attendance: (ticketPrice: Int)Int
```

Write some Tests

Note that we're writing tests *before* we implement the function! We want to think about how the function should work, and write tests to assert that behavior *before* we are biased by the implementation. For `attendance`, we should identify some interesting inputs and expected outputs based on the theater owner's provided data.

In [7]:
```
120 == attendance(500)
```

```
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at attendance(<console>:95)
  ... 46 elided
```

Oops! We haven't implemented `attendence` yet!

Apply the KISS Principle for Definitions

The *KISS* design principle: *"**K**eep **I**t **S**imple, **S**tupid"*

- Write the simplest solution that fits the template and passes the current tests.
- Keeping the definition simple will:
  1. Force us to include adequate test coverage.
  2. Help to keep us from over-engineering.

In [8]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 */
def attendance(ticketPrice: Int): Int = {
    require(ticketPrice >= 0)
120
} ensuring(result => result >= 0)
```

Out[8]:
```
attendance: (ticketPrice: Int)Int
```

In [9]:
```
120 == attendance(500)
```

Out[9]: true

Yay! The test passed—all done!

Iterate until Complete

OK, we were really only done with that iteration of the design recipe.

Let's write more tests, and then update our solution to make them pass.

Iteration 2

Let's try dropping the ticket price by $0.10, which should result in the attendance increasing by 15.

In [10]:
```
135 == attendance(490)
```

Out[10]: false

In [11]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  120 + (500 - ticketPrice) * (15 / 10)
} ensuring(result => result >= 0)
```

Out[11]: attendance: (ticketPrice: Int)Int

Does our original test still work?

In [12]:
```
120 == attendance(500)
```

Out[12]: true

And the new test?

In [13]:
```
135 == attendance(490)
```

Out[13]: false

Agh! What went wrong?! Let's check the actual output.

In [14]:
```
attendance(490)
```

Out[14]: 130

Why didn't we get 135? Let's compute `attendance(490)` by hand.

```
120 + (500 - {490}) * (15 / 10)
↦
  120 + (10) * (15 / 10)
↦
  120 + (10) * (1) // Oops! Integer division truncates...
↦
  120 + 10
↦
  130
```

Let's try multiplying before dividing to avoid this error.

In [15]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  120 + 15 * (500 - ticketPrice) / 10
} ensuring(result => result >= 0)
```

Out[15]: attendance: (ticketPrice: Int)Int

Does our original test still work?

In [16]:
```
120 == attendance(500)
```

Out[16]: true

And the new test?

In [17]:
```
135 == attendance(490)
```

Out[17]: true

Yay! Time to iterate again.

Iteration 3

If we charge too much money, let's say $10, then no one should show up.

Let's try that.

In [18]:
```
0 == attendance(1000)
```

```
java.lang.AssertionError: assertion failed
  at scala.Predef$.assert(Predef.scala:156)
  at scala.Predef$Ensuring$.ensuring$extension2(Predef.scala:260)
  at attendance(<console>:96)
  ... 46 elided
```

Agh, the *postcondition* failed. Let's check the actual result to see why.

In [19]:
```
120 + 15 * (500 - {1000}) / 10
```

Out[19]:
```
-630
```

Yeah, that probably doesn't make sense. 

Scala integers have a built-in `max` method. Let's use that to lower-bound the result.

Note: If there's a useful-looking method or operator available on an object, but we didn't discuss it in class, just make a quick post to Piazza asking if you can use it! We usually say yes unless (1) we want you to implement it on your own, or (2) we think that wanting that method means you're doing something more complex than necessary.

In [20]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance
 */
def attendance(ticketPrice: Int): Int = {
  require(ticketPrice >= 0)
  (120 + 15 * (500 - ticketPrice) / 10).max(0)
} ensuring(result => result >= 0)
```

Out[20]:
```
attendance: (ticketPrice: Int)Int
```

Does our original tests still work?

```
In [21]: 120 == attendance(500)
Out[21]: true
```

```
In [22]: 135 == attendance(490)
Out[22]: true
```

And the new test?

```
In [23]: 0 == attendance(1000)
Out[23]: true
```

Yay!

Iteration 2.1

Let's pretend that Scala didn't have a built-in `max` operator for integers.

The Scala language exposes all of the static methods from `java.lang.Math.*` in the `scala.math` package. Since all members of the `scala` package are imported by default, we can call methods like `math.abs`, `math.min`, and `math.max` directly:

```
In [24]: math.max(-630, 0)
Out[24]: 0
```

However, we haven't defined anything in the `math` package in Core Scala. In fact, we haven't even defined symantics of packages yet, so we probably shouldn't use that either.

Let's implement our own `max` function instead, just like we might do for any non-standard operation we decide to lift into its own function.

At this point we should re-apply our design recipe to this sub-problem.

Analyze

We want to use our `max` function on integers, so the signature should be `(Int, Int)=>Int`.

```
In [25]: def max(x: Int, y: Int): Int = ???
Out[25]: max: (x: Int, y: Int)Int
```

It might also make sense to enforce that the result is one of the inputs.

We should also add our purpose statement as a comment.

In [26]:
```
/** Compute the maximum of two integers */
def max(x: Int, y: Int): Int = {
  ??? : Int
} ensuring(result => result == x | result == y)
```

Out[26]:
```
max: (x: Int, y: Int)Int
```

Write a Test

Let's start with the `max(-630, 0)` case we already had above.

In [27]:
```
0 == max(-630, 0)
```

```
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at max(<console>:90)
  ... 46 elided
```

Update the Definition

Don't go too overboard with the *KISS* principle. We know how `max` should work, and there's no need to iterate on obviously broken implementations like the followng:

In [28]:
```
def max(x: Int, y: Int) = y
```

Out[28]:
```
max: (x: Int, y: Int)Int
```

In [29]:
```
0 == max(-630, 0)
```

Out[29]:
```
true
```

Let's use an `if` expression to check which input is larger, and return that one.

In [30]:
```
/** Compute the maximum of two integers */
def max(x: Int, y: Int): Int = {
  if (x > y) x else y
} ensuring(result => result == x | result == y)
```

Out[30]:
```
max: (x: Int, y: Int)Int
```

In [31]:
```
0 == max(-630, 0)
```

Out[31]:
```
true
```

Yay! Now we have our own Core Scala `max` function!

Iterate

We should *iterate* on the design recipe and add some more test cases...

```
In [32]: 2 == max(1, 2)
```

```
Out[32]: true
```

```
In [33]: Int.MaxValue == max(Int.MaxValue, 0)
```

```
Out[33]: true
```

```
In [34]: Int.MinValue == max(Int.MinValue, Int.MinValue)
```

```
Out[34]: true
```

I'm reasonably confident this is working.

If we were dealing with `Doubles` rather than `Ints` I'd be more worried.

Notice that the actual `math.max` method works for `Doubles`, and can handle `±0.0`:

```
In [35]: math.max(-0.0, +0.0)
```

```
Out[35]: 0.0
```

```
In [36]: math.max(-0.0, -0.0)
```

```
Out[36]: -0.0
```

```
In [37]: math.max(+0.0, -0.0)
```

```
Out[37]: 0.0
```

Sorry for the tangent—let's get back to `attendance`.

Iteration 3

Let's add some more tests. Maybe some edge cases.

```
In [38]: 0 == attendance(Int.MaxValue)
```

```
Out[38]: true
```

That worked! But maybe it shouldn't have...

```
attendance(Int.MaxValue)
↦
attendance(2147483647)
↦
max(0, 120 + 15 * (500 - 2147483647) / 10)  // subtraction overflow!
↦
max(0, 120 + 15 * (-2147483147) / 10)  // multiplication overflow!
↦
max(0, 120 + -2147476133 / 10)
↦
max(0, 120 + -214747613)
↦
max(0, -2147476013)
↦
if (0 >= -2147476013) 0 else -2147476013
↦
if (true) 0 else -2147476013
↦
0
```

Bounding Values Based on Domain Knowledge

We can find appropriate bounds for values by considering the range of those values required by our problem domain. Often, these bounds will be much tighter than those of a generic `Int` or `Double`.

In our example, we can see from our formula that attendance is zero whenever the cost of a ticket is $5.80 or above. We can also see that even free tickets achieve attendance of only 870 people And it is likely that our theater cannot seat 870 people!

Maybe it would make more sense to update our `attendance` contract to further restrict the reasonable values of `ticketPrice`.

In [39]:
```
/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 * Undefined for ticket prices over 1000 cents.
 */
def attendance(ticketPrice: Int): Int = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  (120 + 15 * (500 - ticketPrice) / 10).max(0)
} ensuring(result => result >= 0)
```

Out[39]: attendance: (ticketPrice: Int)Int

Now our `Int.MaxValue` input should be rejected:

```
In [40]: 0 == attendance(Int.MaxValue)
```

```
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:212)
  at attendance(<console>:95)
  ... 46 elided
```

But do the rest of our tests still work?

```
In [41]: 120 == attendance(500)
```

```
Out[41]: true
```

```
In [42]: 135 == attendance(490)
```

```
Out[42]: true
```

```
In [43]: 0 == attendance(1000)
```

```
Out[43]: true
```

Great! Let's add more.

```
In [44]: 0 == attendance(580)
```

```
Out[44]: true
```

```
In [45]: 2 == attendance(579)
```

```
Out[45]: true
```

```
In [46]: 870 == attendance(0)
```

```
Out[46]: true
```

At this point I'm probably pretty confident that `attendance` works.

Keep Applying the Design Recipe

Now we continue applying the design recipe to build a function that solves the next piece of our overall problem. In this case, defining a `cost` function would make sense.

Cost

Applying the design recipe for `cost` might yield the following tests and definition:

In [47]:
```
/**
* Returns cost to the theater of showing a film,
* as a function of ticketPrice.
*/
def cost(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  18000 + 4 * attendance(ticketPrice)
} ensuring(result => result > 0)
```

Out[47]: `cost: (ticketPrice: Int)Int`

In [48]:
```
18420 == cost(510)
```

Out[48]: `true`

In [49]:
```
21480 == cost(0)
```

Out[49]: `true`

In [50]:
```
18000 == cost(1000)
```

Out[50]: `true`

Revenue

In [51]:
```
/**
* Returns revenue received by the theater when
* showing a film, as a function of ticket price.
*/
def revenue(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  ticketPrice * attendance(ticketPrice)
} ensuring(result => result >= 0)
```

Out[51]: `revenue: (ticketPrice: Int)Int`

In [52]:
```
0 == revenue(0)
```

Out[52]: `true`

In [53]:
```
0 == revenue(1000)
```

Out[53]: true

In [54]:
```
53550 == revenue(510)
```

Out[54]: true

Profit

In [55]:
```
/**
* Returns profit enjoyed by the theater after showing
* a film, defined as the difference between revenue
* costs.
*/
def profit(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  revenue(ticketPrice) - cost(ticketPrice)
}
```

Out[55]: profit: (ticketPrice: Int)Int

In [56]:
```
35130 == profit(510)
```

Out[56]: true

In [57]:
```
-21480 == profit(0)
```

Out[57]: true

In [58]:
```
-18000 == profit(1000)
```

Out[58]: true

Would it make sense to add an `ensuring` clause to `profit`?

All Together

In [59]:
```
/** Compute the maximum of two integers */
def max(x: Int, y: Int): Int = {
  if (x > y) x else y
} ensuring(result => result == x | result == y)

/**
 * Given a ticketPrice in cents,
 * returns the number of people expected
 * to attend a performance.
 * Undefined for ticket prices over 1000 cents.
 */
def attendance(ticketPrice: Int): Int = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  (120 + 15 * (500 - ticketPrice) / 10).max(0)
} ensuring(result => result >= 0)

/**
* Returns cost to the theater of showing a film,
* as a function of ticketPrice.
*/
def cost(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  18000 + 4 * attendance(ticketPrice)
} ensuring(result => result > 0)

/**
* Returns revenue received by the theater when
* showing a film, as a function of ticket price.
*/
def revenue(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  ticketPrice * attendance(ticketPrice)
} ensuring(result => result >= 0)

/**
* Returns profit enjoyed by the theater after showing
* a film, defined as the difference between revenue
* costs.
*/
def profit(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  revenue(ticketPrice) - cost(ticketPrice)
}
```

Out[59]:
```
max: (x: Int, y: Int)Int
attendance: (ticketPrice: Int)Int
cost: (ticketPrice: Int)Int
revenue: (ticketPrice: Int)Int
profit: (ticketPrice: Int)Int
```

Note that we can inline all of this logic into the `profit` function (excluding `max` to avoid too much repeated code):

In [60]:
```
def profit(ticketPrice: Int) = {
  require(0 <= ticketPrice & ticketPrice <= 1000)
  ticketPrice * max(0, 120 + 15 * (500 - ticketPrice) / 10) -
    18000 + 4 * max(0, 120 + 15 * (500 - ticketPrice) / 10)
}
```

Out[60]:
```
profit: (ticketPrice: Int)Int
```

Which version would you rather have when you need to adjust the numbers 5 years later due to inflation?

How Many Functions Do I Need?

As a guideline:

- Include a helper function for each of the dependencies mentioned in your problem statement.
- Include a helper function for new dependencies discovered during testing.