Object-Oriented Programming in Smalltalk

COMP 105 Assignment

Due Tuesday, May 2, 2017 at 11:59PM

Contents

Overview

The purpose of this assignment is to help you get acquainted with pure object-oriented programming. Even simple algorithms tend to be implemented by sending lots of messages back and forth among a cluster of cooperating, communicating objects. The assignment is divided into two parts.

- In the first part, you do a few small warmup problems to get you used to pure object-oriented style and to acquaint you with μSmalltalk's large initial basis.
- In the second part, you implement *bignums* in μSmalltalk. Bignums are useful in their own right, and they show how to use *double dispatch* when it is needed to expose representation selectively. You will go beyond the simple natural numbers of the SML assignment to include large signed integers and "mixed arithmetic" of large and small integers. Such mixed arithmetic cannot easily be achieved using abstract data types.

This assignment is unusually time-consuming. Many students have experience in languages called object-oriented, but very few students have experience with the extensive, pervasive inheritance that characterizes idiomatic Smalltalk programs.

Setup

The μSmalltalk interpreter is in `/comp/105/bin/usmalltalk`. Many useful μSmalltalk sources are included the book's git repository, which you can clone by

```
git clone homework.cs.tufts.edu:/comp/105/build-prove-compare
```

Sources that can be found in the `examples` directory includes copies of predefined classes, collection classes, financial history, and other examples from the textbook.

Interactive debugging with μSmalltalk

Smalltalk is a small language with a big initial basis. There are lots of predefined classes and methods. Here are two tools that can help you get oriented and debug your code:

- Every *class* understands the messages `protocol` and `localProtocol`. These methods are shown in Figure 11.7 on page 881. They are a quick way to remind yourself what messages an object understands and how the message names are spelled.
- The interpreter can help you debug by emiting a *trace* of message sends and answers to depth `n`. Just enter the definition

  ```
  (val &trace n)
  ```

  at the read-eval-print loop.

Reading comprehension (10 percent)

These problems will help guide you through the reading. We recommend that you complete them before starting the other problems below. You can download the questions[1].

1. Read the first ten pages of chapter 11.

   (a) What is a "receiver"? What is an "argument"? What is the difference?

   (b) What is the relationship between a *message* and a *protocol*?

   (c) What is the difference between *instance* protocol and *class* protocol?

   (d) In the name of a message, what is the significance of a colon or colons?

   You are ready for exercise 9 (collection protocols).

2. Read Section 11.3.4 on message send and dynamic dispatch. Using the class definitions in that section, message `m1` is sent to an object of class `C`: What method *definitions* are evaluated in what order? An answer looks something like "method m5 on class H, then method m12 on class E, …".

3. Continuing with method dispatch, now study the *class* method `new` defined on class `List` in code chunk 927a. The definition sends message `new` to `super`. (Keep in mind: because `new` is a *class* method, both `super` and `self` stand for the class, not for any instance.)

   (a) When *class* method `new` is evaluated, what three messages are sent by the method body, in what order?

   (b) What does each message send accomplish?

   (c) If we changed the body so instead of `(new super)` it said `(new self)`, what would happen?

4. Find the implementation of class `Number` on pages 933 and 934. Now study the implementation of class `Fraction` pages 936 to 938.

   Message - (minus) is sent to `Fraction (/ 1 2)` with argument `Fraction (/ 1 3)`. What methods are evaluated in what order? Which class implements `-`?

   You are getting ready for exercise 30(a).

5. Read about the `Magnitude` protocol on pages 894 and 895. Study the implementation of class `Magnitude` on page 932. Then study the implementations of classes `Integer` and `SmallInteger` on pages 934 and 935.

   Message <= is sent to small integer 3 with argument 9. What methods are evaluated in what order? Which class implements <=?

   You are ready for exercise 30(a), and you are ready to implement class `Natural` (exercise 32).

6. In section 11.10.1 on page 978, you will find a short definition of an "abstract class." What do you think is the purpose of an abstract class?

7. One example of an abstract class is class `Number`. Look up its definition on page 933. List all the methods that are "subclass responsibility." These are the methods you must implement in class `LargeInteger`.

[1] ./cqs.small.txt

You are getting ready to implement large integers.

8. Study the implementation of common methods on classes `Magnitude` and `Number`. This question asks you to apply the ideas to `LargeInteger`: Of the methods listed in the previous comprehension question (subclass responsibilities of `Number`), which ones can be implemented on class `LargeInteger` itself, as opposed to one of its concrete subclasses? (None of these methods will require double dispatch, which is the subject of the next comprehension question.)

   You are ready to write a few simple methods for Exercise 33 (large integers).

9. Read Section 11.6.4 on pages 931 and 932.

   Of the methods listed in comprehension question 7 (subclass responsibilities of `Number`), list each one that needs to know whether its *argument* (not the receiver) is a large or small integer, and if large, whether it is positive or negative. These methods will require double dispatch.

   (a) List the methods. For example, `+` is such a method.

   (b) For each method, list what method it will dispatch to on its argument if the received is a `LargePositiveInteger`. For example, method `+` will dispatch to `addLargePositiveIntegerTo:`.

   You are ready to implement large integers (Exercise 33).

10. Read about coercion in section Section 11.4.5 on page 895. Look at the protocol on page 894.

    Explain the roles of the methods `asInteger`, `asFraction`, `asFloat`, and `coerce:`. If you are unsure, look at the implementations of these methods on class `Integer` on pages 934 and 935.

    You are ready to implement mixed arithmetic, with coercions, in Exercise 34.

Individual Problems

Working on your own, please solve Exercise 9 on page 986 and Exercise 30(a) on page 991 of Ramsey. These exercises are warm-ups designed to prepare you for the Bignum problems in the pair portion of the assignment.

9. *Collection Protocols*. Do Exercise 9 on page 986 of Ramsey.

Related reading: Section 11.1, which uses the FinancialHistory class as the running example.

30(a). *Interfaces as Abstraction Barriers*. Do Exercise 30(a) on page 991 of Ramsey. When the problem says “Arrange the `Fraction` and `Integer` classes”, the text means to revise one or both of these classes or define a related class. Think about *protocols*, not implementation.

Put your solution in file `frac-and-int.smt`. At minimum, your solution should support addition, subtraction, multiplication, so include at least one unit test for each of these operations.

Hints:

- In a system with abstract data types, you can’t easily mix integers and fractions; they have different types. But in an object-oriented system with behavioral subtyping, you just have to get one object to “behave like” another—which means implementing its protocol. In some cases, this might include implementing private methods.

- If you change `Integer`, this change doesn't affect `SmallInteger`, which continues to inherit from the original version of `Integer`. So if you change `Integer`, count on changing `SmallInteger` as well.

Related reading:

- For an overview of the `Magnitude` class and its relationship to numbers, read the first page of Section 11.4.5, which starts on page 895. Also in that section, read about `Integer` and `Fraction`.
- For the implementation of `Integer`, read pages 934 and 935. For the time being, you can ignore class `SmallInteger`.
- For the implementation of `Fraction`, read pages 936 to 938. Study the implementation of method `+`, and observe how it relies on the exposure of representation through private methods `num` and `den`.
- If nothing comes to you, try reading about how we get access to multiple representations in the object-oriented way: Section 11.6.4 on pages 931 and 932. You will need to read this section later anyway.

How big is it? To solve these problems, you shouldn't need to add or change more than 10 lines of code in total. The optimal solution is no more than a few lines long.

Pair Problems: Bignum arithmetic

For these problems, you may work with a partner. Please solve Exercise 32 on page 991, Exercise 33 on page 994, and Exercise 34 on page 994 of Ramsey, and Exercise **T** below. You do not need to implement long division.

Sometimes you want to do computations that require more precision than you have available in a machine word. Full Scheme, Smalltalk, and Icon all provide "bignums." These are integer implementations that automatically expand to as much precision as you need. Because of their dynamic-typing discipline, these languages make the transition transparently—you can't easily tell when you're using native machine integers and when you're using bignums. In this portion of the homework, you will reimplement infinite-precision natural numbers in Smalltalk, use these natural numbers to define infinite-precision integers, and then modify the built-in `SmallInteger` class so that when more precision is needed, operations seamlessly roll over to infinite precision.

Big picture of the solution

Smalltalk code sends lots of messages back and forth among lots of methods that are defined on different classes. This model shows both the power of Smalltalk—you get a lot of leverage and code reuse—and the weakness of Smalltalk—every algorithm is smeared out over half a dozen methods defined on different classes, making it hard to debug. But this is the object-oriented programming model that lends power not just to Smalltalk but also to Ruby, Objective-C, Swift, Self, JavaScript, and to a lesser extent, Java and C++. To work effectively in any of these languages, one needs the big picture of which class is doing what. A good starting point is the Smalltalk bignums handout[2]

[2] ../handouts/bignums.pdf

Unit testing bignums in Smalltalk

Like the other bridge languages, μSmalltalk has `check-expect`. But in Smalltalk, checking for equality is often not useful—because many Smalltalk abstractions are mutable, equality is normally defined as *object identity*. For example, two lists compare equal only if they are the same list. If they are different lists containing the same elements, then the = method says they are different.

To test equality of bignums in Smalltalk, we will convert each bignum to a list of digits, which we will compare for similarity using the `has:to:` *class* method on class `Similarity`:

```
(class Similarity Object
  ()
  (class-method has:to: (ms ns)
    (locals i n equivalent)
    (set n (size ms))
    (if (!= n (size ns))
        {false}
        {(set i 1)
         (set equivalent true)
         (while {(and: equivalent {(<= i n)})}
            {(ifTrue:ifFalse: (!= (at: ms i) (at: ns i))
                {(set equivalent false)}
                {(set i (+ i 1))})})
         equivalent})))
```

With this class method, we can even compare a list and an array for similarity, successfully. Here's an example of how it works.

```
-> (val ns (new List))
List( )
-> (add: ns 1)
-> (add: ns 2)
-> (add: ns 3)
-> ns
List( 1 2 3 )
-> (= ns ns)
<True>
-> (= ns #( 1 2 3 ))
<False>
-> (has:to: Similarity ns #( 1 2 3 ))
<True>
```

If you want to test the results of *arithmetic*, your `check-expect` should send message `has:to:` to class `Similarity` and will compare the result with `true`. You can download class `Similarity`[3]. If you want to test *comparisons*, your `check-expect` can simply refer to `true` and `false` as usual.

[3] ./similarity.smt

How to prepare your code for our testing

Many of our tests interact directly with the μSmalltalk interpreter. Our testing infrastructure enters definitions and looks at the responses. To pass our stringent test suites, *you must define* `print` *methods that render numbers as normal people expect to see them*. You cannot simply send `decimal` to `self` and print the result—you must print the individual digits, possibly preceded by a minus sign.

Array alert

Smalltalk arrays are numbered from 1, not from 0. It is *very* annoying that you are stuck with a 1-based array type to implement a 0-based abstraction (polynomials) for class `Natural`. The book recommends private methods `digit:` and `digit:put:`. I recommend that you define these methods, which **hide** the 1-based nature of the underlying representation:

```
(method digit: (i)
   (at:ifAbsent: digits (+ i 1) {0}))
(method digit:put: (i d)
   (at:put: digits (+ i 1) d))
```

Details of all the problems

32. *Implementing arbitrary precision natural numbers*. Do Exercise 32 on page 991 of Ramsey. **Modify the protocol given in the assignment** according to these instructions:

- As in the previous assignment, your `div:` method need only divide a `Natural` by a small integer, not by another `Natural`.
- Add to the protocol the method `decimal` described in Figure 11.21. This method answers a list of decimal digits that represent the number. As in the previous assignment, the `decimal` method will be used for testing.

The algorithms are the same algorithms you would have used on the `sml` assignment[4]. And although the book assumes you will represent a natural number using an array, you are free to use a list of digits instead. Just be aware that unlike Standard ML, μSmalltalk has no *immutable* list abstraction. If you want immutable lists, you will have to simulate them yourself. For example, you could try using the `Cons` and `Nil` classes that are already defined, along with the `isNil` message that is defined on every object.

Please do adapt your code from the `sml` assignment. Or if you prefer, **you may adapt my solutions.** Just be sure to attribute the source for whatever code you adapt.

How big is it? My solution for the `Natural` class, using the array representation and the hints in the book, is over 100 lines of μSmalltalk code.

Related reading:

- There is a detailed implementation guide in the bignums handout[5].

[4] sml.html
[5] ../handouts/bignums.pdf

- In a system with abstract data types, binary operations like `+` and `*` are easy: you automatically have access to both representations. In a system with objects, not so! To learn how to get access to multiple representations in the object-oriented way, read section 11.6.4 on pages 931 and 932.
- Class `Natural` is a subclass of `Magnitude`. Study the `Magnitude` protocol in Section 11.4.5. For information about the implementation of `Magnitude`, which should provide useful ideas about `Natural`, as well as the "subclass responsibilities," study the implementation of `Magnitude` in section 11.6.5 on page 932.[6]
- For the interface to a Smalltalk array, study the `Collection` protocol in Section 11.4.4, which starts on page 884. You have access to the protocol in Figure 11.9, but you are more likely to use the `KeyedCollection` protocol in Figure 11.10, especially `at:` and `at:put:`. Don't overlook the **Arrays** section on pages 892 and 893, including its description of the array *class* methods `new:` and `from:`.
- For list construction, which you will need for the `decimal` method, look at the `List` protocol in Section 11.4.4, especially Figure 11.12.

33. *Implementing arbitrary precision integers*. Do Exercise 33 on page 994 of Ramsey. You need not implement the `div:` method. But in order to facilitate testing, **you must implement a `decimal` method** like the one you implemented on class `Natural`. An implementation is shown in the bignums handout[7].

Because you build large integers on top of `Natural`, you don't have to think about array or list representations any more. Instead you must focus on dynamic dispatch and on getting information from where it is to where it is needed.

How big is it? My solutions for the large integer classes are 22 lines apiece.

Related reading: This problem is all about dynamic dispatch, including double dispatch. Read Section 11.6.4, which starts on page 931. (You'll also have a chance to practice double dispatch in recitation.)

34. *Modifying `SmallInteger` so operations that overflow roll over to infinite precision*. Do Exercise 34 on page 994 of Ramsey.

You must modify `SmallInteger` *without* editing the source code of the μSmalltalk interpreter. To do so, you will *redefine* class `SmallInteger` using the idiom on page 995:

```
(class SmallInteger SmallInteger
   ()
   ... new method definitions ...
)
```

This idiom modifies the existing class `SmallInteger`; it can both change existing methods and define new methods. **This code changes the basic arithmetic operations that *the system uses internally*.** If you have bugs in your code, the system will behave erratically. At this point, you must restart your interpreter and fix your bugs. Then use the idiom again.

How big is it? My modifications to the `SmallInteger` class are about 25 lines.

[6]Note: an object of class `Natural` is not a `Number` as Smalltalk understands it. In particular, class `Natural` does not support methods `negated` or `reciprocal`.

[7]../handouts/bignums.pdf

Related reading: Everything about dispatch and double dispatch still applies. In addition, you need to know how overflow is handled using "exception blocks."

- Review the presentation of blocks, especially the *parameterless* blocks (written with curly braces) in Section 11.4.3, which starts on page 883.
- A very simple example of an exception block appears above in the recommended implementation of method `digit:`. Read the description of `at:ifAbsent:` in the keyed-collection protocol in Figure 11.10 on page 890. Now study the example method:

  ```
  (method digit: (i)
     (at:ifAbsent: digits (+ i 1) {0}))
  ```

  Any reference out of bounds sends `value` to the "exception block" `{0}`, which answers zero.
- Study the implementation of the `at:` method in chunk 923d, which uses `at:ifAbsent:` with an "exception block" that causes a run-time error if `value` is sent to it.
- Finally, study the overflow-detecting primitive methods at the top of page 995, and study the implementation of `addSmallIntegerTo:` in code chunk 995a. That is the technique you must emulate.

T. *Testing Bignums*. You will write 9 tests for bignums:

- 3 tests will test only class `Natural`.
- 3 tests will test the large-integer classes, which are built on top of class `Natural`.
- 3 tests will test mixed arithmetic and comparison involving both small and large integers.

Each test will have the same structure:

1. The test will begin with a **summary characterization** of the test in at most 60 characters, formatted on a line by itself as follows:

   ```
      ; Summary: .........
   ```

   The summary must be a simple English phrase that describes the test. Examples might be "Ackermann's function of (1, 1)," "sequence of powers of 2," or "combinations of +, *, and - on random numbers."
2. Code will compute a result of class `Natural`, `LargePositiveInteger`, or `LargeNegativeInteger`. The code may appear in a method, a class method, a block, or wherever else you find convenient.
3. The expected result will be written as a literal array, like one of the following:

   ```
   #( 1 2 3 9 4 9 )
   #( - 6 5 5 3 6 )
   ```
4. The test itself will send the `decimal` message to the computed result, then compare the decimal representation with the literal array. Comparison will use class method `has:to:` on class `Similarity`.
5. Finally, the comparison will appear in a `check-expect`, expecting the result of `has:to:` to be `true`.[8]

[8] In μSmalltalk, truth is `true`, not `#t`.

Each test must take less than 2 CPU seconds to evaluate.

Here is a complete example:

```
; Summary: 10 to the tenth power, mixed arithmetic
(class Test10Power Object
  ()
  (class-method run: (power)
     [locals n 10-to-the-n]
     (set n 0)
     (set 10-to-the-n 1)
     (whileTrue: {(< n power)}
         {(set n (+ n 1))
          (set 10-to-the-n (* 10 10-to-the-n))})
     10-to-the-n)
)
(check-expect (has:to: Similarity (decimal (run: Test10Power 10))
                                  #( 1 0 0 0 0 0 0 0 0 0 0))
              true)
```

Here is another complete example:

```
; Summary: 20 factorial
(define factorial (n)
  (ifTrue:ifFalse: (strictlyPositive n)
     {(* n (value factorial (- n 1)))}
     {1}))

(check-expect (has:to: Similarity (decimal (value factorial 20))
                                  #( 2 4 3 2 9 0 2 0 0 8 1 7 6 6 4 0 0 0 0 ))
              true)
```

Related reading: No special reading is recommended for the testing problem. As long as you understand the examples above, that should be enough.

A simple sanity check

As a test, the factorial function has grave limitations:

- Factorial only ever multiplies numbers. It does not add, subtract, negate, or compare numbers.
- Factorial never multiplies two large numbers. It only ever multiplies a large number by a small number, or two small numbers.

These properties of factorial make it a poor test of correctness, but they do make it a good initial sanity check. Here, for example, is code that computes and prints factorials[9]:

```
(class Factorial Object
  ()
  (class-method printUpto: (limit) [locals n nfac]
```

[9] ./factorial.smt

```
(set n 1)
(set nfac 1)
(whileTrue: {(<= n limit)}
   {(print n) (print #!) (printu 32) (print #=) (printu 32) (println nfac)
    (set n (+ n 1))
    (set nfac (* n nfac))})))
```

As a sanity check sending `(printUpto: Factorial 25)` should print the following table of factorials:

```
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884976640000
24! = 620448401733239439360000
25! = 15511210043330985984000000
```

More advice about testing natural numbers

Try testing your class `Natural` by generating a long, random string of digits, then computing the corresponding number using a combination of addition and multiplication by 10.

If you don't have multiplication working yet, you can use the following sequence to multiply by 10:

```
(set p n)       ; p == n
(set p (+ p p)) ; p == 2n
(set p (+ p p)) ; p == 4n
(set p (+ p n)) ; p == 5n
(set p (+ p p)) ; p == 10n
```

This idea will test only your addition; if you have bugs there, fix them before you go on.

You can write, in μSmalltalk itself, a method that uses the techniques above to convert a sequenceable collection of decimal digits into a natural number.

Once you are confident that addition works, you can test subtraction of natural numbers by generating a long random sequence, then subtracting the same sequence in which all digits except the most significant are replaced by zero.

You can create more ambitious tests of subtraction by generating random natural numbers and using the algebraic law $(m + n) - m = n$. You can also check to see that unless n is zero, $m - (m + n)$ causes a run-time error on class `Natural`.

It is harder to test multiplication, but you can at least use repeated addition to test multiplication by *small* values. The `timesRepeat:` method is defined on any integer.

You can also easily test multiplication by large powers of 10.

You can use similar techniques to test large integers.

If you want more effective tests of multiplication and so on, compare your results with a working implementation of bignums. The languages Scheme, Icon, and Haskell all provide such implementations. (Be aware that the real Scheme `define` syntax is slightly different from what we use in uScheme.) We recommend you use `ghci` on the command line; standard infix syntax works. If you want something more elaborate, use Standard ML of New Jersey (command `sml`), which has an `IntInf` module that implements bignums.

Other hints and guidelines

Start early. Seamless arithmetic requires in-depth cooperation among about eight different classes (those you write, plus `Magnitude`, `Number`, `Integer`, and `SmallInteger`). This kind of cooperation requires aggressive message passing and inheritance, which you are just learning. There is a handout online[10] with suggestions about which methods depend on which other methods and in what order to tackle them.

For your reference, Dave Hanson's book[11] discusses bignums and bignum algorithms at some length. It should be free online to Tufts students. You can think about borrowing code from Hanson's implementation[12] (see also his distribution[13]). Be aware though that your assignment differs significantly from his code and unless you have read the relevant portions of the book, you may find the code overwhelming.

- In Hanson's code, `XP_add` does add with carry. `XP_sub` does subtract with borrow. `XP_mul` does `z := z + x * y`, which is useful, but is not what we want unless `z` is zero initially.
- Hanson passes all the lengths explicitly, which would not be idiomatic in μSmalltalk.
- Hanson's implementation uses mutation extensively, but the class `Natural` is an immutable type. Your methods must *not* mutate existing natural numbers; you can mutate only a newly allocated number that you are sure has not been seen by any client.

If you do emulate Hanson's code, acknowledge him in your README file.

[10] ../handouts/bignums.pdf
[11] ../readings/indexbody.html#cii
[12] xp.c
[13] http://www.cs.princeton.edu/software/cii

Avoid common mistakes

Below you will find some common mistakes to avoid.

It is common to overlook class methods. They are a good place to put information that doesn't change over the life of your program.

It's not common, but it's a **terrible mistake** to try to implement mixed operations by interrogating an object about its class—a so-called "run-time type test." Run-time type tests destroy behavioral subtyping. If you find yourself wanting to ask an object what its class is, seek help immediately.

It is surprisingly common for students to submit code for the small problems without ever even having run the code or loaded it into an interpreter. If you run even one test case, you will be ahead of the game.

It is too common to submit bignum code without having tested all combinations of methods and arguments. Your best plan is to write a program, in the language if your choice, that loops over operator and both operands and generates at least one test case for every combination. Because μSmalltalk is written using S-expressions, you could consider writing this program in μScheme—but any language will do.

It is relatively common for students' code to make a false distinction between two flavors of zero. In integer arithmetic, there is only one zero, and it always prints as "`0`".

It's surprisingly common to fail to tag the test summary with the prefix `Summary:`, or to forget it altogether.

Extra credit

Seamless bignum arithmetic is an accomplishment. But it's a long way from industrial. The extra-credit problems explore some ideas you would deploy if you wanted everything on a more solid foundation.

Base variations. A key problem in the representation of integers is the choice of the base b. Today's hardware supports `b = 2` and sometimes `b = 10`, but when we want bignums, the choice of `b` is hard to make in the general case:

- If `b` = 10, then converting to decimal representation is trivial, but storing bignums requires lots of memory.
- The larger `b` is, the less memory is required, and the more efficient everything is.
- If `b` is a power of 10, converting to decimal is relatively easy and is very efficient. Otherwise, as long as $b \geq 10$, conversion requires only short division.
- If `(b-1) * (b-1)` fits in a machine word, than you can implement multiplication in high-level languages without difficulty. (Serious implementations pick the largest `b` such that one digit fits in a machine word, e.g., 2^{64} on modern machines. Unfortunately, to work with such large values of `b` requires special machine instructions to support "add with carry" and 128-bit multiply, so serious implementations have to be written in assembly language.)
- If `b` is a power of 2, bit-shift can be very efficient, but conversion to decimal is expensive. Fast bit-shift can be important in cryptographic and communications applications.

If you want signed integers, there are more choices: signed-magnitude and `b`'s-complement. Knuth volume 2[14] is pretty informative about these topics.

For extra credit, try the following variations on your implementation of class `Natural`:

1. Implement the class using an internal base $b = 10$. Measure the time needed to compute the first 50 factorials.
2. Argue for the largest possible base that is still a power of 10. Change your class to use that base internally. (If you are both careful and clever, you should be able to change only the class method `base` and not any other code.) Measure the time needed to compute **and print** the first 50 factorials.
3. Argue for the largest possible base that is a power of 2. Change your class to use that base internally. Measure the time needed to compute **and print** the first 50 factorials. Does having fewer digits recoup the higher cost of converting to decimal?

Because Smalltalk hides the representation from clients, a well-behaved client won't be affected by a change of base. If we wanted, we could take more serious measurements and pick the most efficient representation.

Remember that the private `decimal` method must return a list of **decimal** digits, even if base 10 is not what is used in the representation. Suppress leading zeroes unless the value of `Natural` is itself zero.

Write up your arguments and your measurements in your README file.

Long division. Implement long division for `Natural` and for large integers. If this changes your argument for the largest possible base, explain how. This article[15] by Per Brinch Hansen describes how to implement long division.

Mixed Comparisons. Make sure comparisons work, even with mixed kinds of integers. So for example, make sure comparisons such as `(< 5 (* 1000000 1000000))` produce sensible answers.

Space costs. Instrument your `Natural` class to keep track of the size of numbers, and measure the space cost of the different bases. Estimate the difference in garbage-collection overhead for computing with the different bases, given a fixed-size heap.

Pi (hard). Use a power series to compute the first 100 digits of pi (the ratio of a circle's circumference to its diameter). Be sure to cite your sources for the proper series approximation and its convergence properties. *Hint: I vaguely remember that there's a faster convergence for pi over 4. Check with a numerical analyst.*

What and how to submit: Individual problems

Submit these files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - The numbers of the problems that you solved

[14] ../readings/indexbody.html#knuth2
[15] ../readings/indexbody.html#division

    - The number of hours you worked on the assignment.
- A file `cqs.small.txt` containing your answers to the reading-comprehension questions
- A file `finhist.smt` showing your solution to Exercise 9.
- A file `frac-and-int.smt` showing whatever definitions you used to do Exercise 30(a). It probably includes new definitions (or redefinitions) of one or more of these classes: `Fraction`, `Integer`, and `SmallInteger`. And it most definitely includes at least four unit tests.

Please identify your solutions using *conspicuous* comments, e.g.,

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;  Solution to Exercise 9
(class Array ...
 )
```

As soon as you have the files listed above, run `submit105-small-solo` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

What and how to submit: Pair problems

Submit these files:

- A `README` file containing
    - A description of how you tested your bignum code
    - The names of the people with whom you collaborated
    - The numbers of the problems that you solved (including any extra credit)
    - Narrative and measurements to accompany your extra-credit answers, if any
- A file `bignum.smt` showing your solutions to Exercises 32, 33, and
    34. This file **must** work with an *unmodified* `usmalltalk` interpreter. Therefore if you use results from Exercise 30(a), or any other problem, you will need to duplicate those modifications in `bignum.smt`.
- A file `bigtests.smt` containing your solution to Exercise T.

As soon as you have the files listed above, run `submit105-small-pair` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

How your work will be evaluated

All our usual expections for **form**, **naming**, and **documentation** apply. But in this assignment we will focus on **clarity** and **structure**. To start, we want to be able to understand your code.

	Exemplary	Satisfactory	Must Improve
Clarity	• Course staff see no more code than is needed to solve the problem. • Course staff see how the structure of the code follows from the structure of the problem.	• Course staff see somewhat more code than is needed to solve the problem. • Course staff can relate the structure of the code to the structure of the problem, but there are parts they don’t understand.	• Course staff roughly twice as much code as is needed to solve the problem. • Course staff cannot follow the code and relate its structure to the structure of the problem.

Structurally, your code should hide information like the base of natural numbers, and it should use proper method dispatch, not bogus techniques like run-time type checking.

	Exemplary	Satisfactory	Must Improve
Structure	• The base used for natural numbers appears in exactly one place, and all code that depends on it consults that place. • *Or*, the base used for natural numbers appears in exactly one place, and code that depends on either consults that place or assumes that the base is some power of 10 • No matter how many bits are used to represent a machine integer, overflow is detected by using appropriate primitive methods, not by comparing against particular integers. • Code uses method dispatch instead of conditionals. • Mixed operations on different classes of numbers are implemented using double dispatch. • *Or*, mixed operations on different classes of numbers are implemented by arranging for the classes to share a common protocol. • *Or*, mixed operations on different classes of numbers are implemented by arranging for unconditional coercions. • Code deals with exceptional or unusual conditions by passing a suitable `exnBlock` or other block. • Code achieves new functionality by reusing existing methods, e.g., by sending messages to `super`. • *Or*, code achieves new functionality by adding new methods to old classes to respond to an existing protocol. • An object's behavior is controlled by dispatching (or double dispatching) to	• The base used for natural numbers appears in exactly one place, but code that depends on it knows what it is, and that code will break if the base is changed in any way. • Overflow is detected only by assuming the number of bits used to represent a machine integer, but the number of bits is explicit in the code. • Code contains one avoidable conditional. • Mixed operations on different classes of integers involve explicit conditionals. • Code protects itself against exceptional or unusual conditions by using Booleans. • Code contains methods that appear to have been copied and modified. • An object's behavior is influenced by interrogating it to learn something about its class.	• The base used for natural numbers appears in multiple places. • Overflow is detected only by assuming the number of bits used to represent a machine integer, and the number of bits is *implicit* in the value of some frightening decimal literal. • Code contains more than one avoidable conditional. • Mixed operations on different classes of integers are implemented by interrogating objects about their classes. • Code copies methods instead of arranging to invoke the originals. • Code contains case analysis or a conditional that depends on the class of an object.

Exemplary	Satisfactory	Must Improve