Object-Oriented Programming in Smalltalk

COMP 105 Assignment

Due Tuesday, April 30, 2019 at 11:59PM

Contents

Overview

Object-oriented programming has been popular since the 1990s, and like lambdas, object-oriented features are found everywhere. But these features are not always easy to tease out: many object-oriented languages, such as Java and C++, are *hybrids*, which mix objects with abstract data types or other notions of encapsulation and modularity. When you don't already know how to program with objects, hybrid designs are more confusing than helpful. For that reason, we study *pure* objects, as popularized by Smalltalk: even simple algorithms send lots of messages back and forth among a cluster of cooperating, communicating objects. Popular languages that use similar models include Ruby, JavaScript, and Objective C.

The assignment is divided into three parts.

- You begin with reading comprehension.
- You do a small warmup problem, which acquaints you with pure object-oriented style and with μSmalltalk's large initial basis.
- You implement *bignums* in μSmalltalk. As in the SML assignment, you will implement both natural numbers and signed integers. You will also use object-oriented dispatch to implement "mixed arithmetic" of large and small integers—a useful abstraction that demonstrates the "open" nature of true object-oriented systems.

This assignment is time-consuming. Many students have experience in languages called "object-oriented," but few students have experience with the extensive inheritance and pervasive dynamic dispatch that characterize 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, shape classes, and other examples from the textbook.

Getting to know μSmalltalk interactively

Smalltalk is a little language with a big initial basis; there are lots of predefined classes and methods. To help you work with the big basis, as well as to debug your own code, we recommend two tools:

- Every *class* understands the messages `protocol` and `localProtocol`. These methods are shown in Figure 10.8 on page 826. They provide a quick way to remind yourself what messages an object understands and how the message names are spelled.

  To learn a protocol the lazy person's way, send an object every message in its protocol, just to see what happens. You might get a stack trace, but that will help you learn.

- The interpreter can help you debug by emitting a *call trace* of up to `n` message sends and answers. Just enter the definition

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

at the read-eval-print loop. To turn tracing off, `(set &trace 0)`.

Here is an (abbreviated) example trace of message `new` sent to class `List`, which is the subject of one of the reading-comprehension questions:

```
-> (new List)
standard input, line 3: Sending message (new) to class List
  predefined, line 478: Sending message (new) to class SequenceableCollection
  predefined, line 478: (new <class List>) = <List>
  predefined, line 478: Sending message (new) to class ListSentinel
    predefined, line 469: Sending message (new) to class Cons
    predefined, line 469: (new <class ListSentinel>) = <ListSentinel>
   predefined, line 470: Sending message (pred: <ListSentinel>) to an object of class ListSentinel
    predefined, line 470: (pred: <ListSentinel> <ListSentinel>) = <ListSentinel>
   predefined, line 471: Sending message (cdr: <ListSentinel>) to an object of class ListSentinel
    predefined, line 471: (cdr: <ListSentinel> <ListSentinel>) = <ListSentinel>
  predefined, line 478: (new <class ListSentinel>) = <ListSentinel>
 predefined, line 478: Sending message (sentinel: <ListSentinel>) to an object of class List
  predefined, line 478: (sentinel: <List> <ListSentinel>) = <List>
standard input, line 3: (new <class List>) = <List>
<trace ends>
List( )
```

One warning: as usual, the interpreter responds to an expression evaluation by printing the result. But in Smalltalk, printing is achieved by sending the `println` message to the result object. This interaction is *also* traced, and the results can be startling. Here, for example, is the result of evaluating the literal integer 3:

```
-> 3
internally generated SEND node, line 1: Sending message (println) to object of class SmallInteger
 internal expression "(begin (print self) (print newline) self)", line 1: Sending message (print) to objec
3  internal expression "(begin (print self) (print newline) self)", line 1: (print 3<SmallInteger>) = 3<Sma
 internal expression "(begin (print self) (print newline) self)", line 1: Sending message (print) to objec
  predefined classes, line 229: Sending message (printu) to object of class SmallInteger

   predefined classes, line 229: (printu 10<SmallInteger>) = 10<SmallInteger>
 internal expression "(begin (print self) (print newline) self)", line 1: (print <Char>) = 10<SmallInteger
internally generated SEND node, line 1: (println 3<SmallInteger>) = 3<SmallInteger>
```

As you see, even a simple operation like printing a number involves *four* message sends. Don't let them confuse you.

Analyzing your code for potential faults

Smalltalk has no type system. But you can still check some properties of your code.

- A "wrong number of arguments" check is built into the language. A symbolic message like `+` or `!=` expects exactly one argument, plus the receiver. An alphanumeric message like `println` or `ifTrue:ifFalse:` expects a number of arguments equal to the number of colons in the message's name—plus the receiver. If these expectations aren't met, the offending code is flagged with a syntax error.

- We provide a simple static-analysis tool called `lint-usmalltalk`. It checks each message send to be sure a method with that name is defined *somewhere*. If no such method is defined, the message name is misspelled.

  Normally, `lint-usmalltalk` also checks for unused methods. An unused method might be misspelled, or it might just be one that isn't used in any code or test. Here's an example run:

  ```
  % lint-usmalltalk bignum.smt
  instance method compare: of class Natural is never used anywhere
  instance method smod: of class Natural is never used anywhere
  ```

- There is a static analysis you can do yourself called *call-graph* analysis. It can help you understand how classes work together, and it's the best tool for diagnosing problems with infinite loops and recursions.

  Call-graph analysis works by identifying what methods transfer control to what other methods. Every *node* in the call graph is a combination of *class name* and *message name*. For example, `Boolean/ifTrue:` or `List/select:`. Each node has outgoing edges that illustrate what happens if the node's message is sent to an instance of the node's class:[1]

  1. If the message is implemented by a *primitive method*, like `+` on `SmallInteger`, it has no outgoing edges.
  2. If the method is *inherited from the superclass*, the node has a *dotted* edge to the same message on the superclass. For example, node `True/ifTrue:` has a dotted edge to `Boolean/ifTrue:`.
  3. If the method is a *subclass responsibility*, the node has a *dotted* edge to *each* subclass.
  4. If the method is *defined* on the class, the node has a *solid* outgoing edge for each message that could be sent during the method's execution.

     You have to look at each message send and figure out what class of object it might be sent to. This part can't easily be determined by looking at the code; you have to know the protocol. For example, if message `&` is sent to a `Boolean`, we know the argument is also a `Boolean`. As another example, if message `+` is sent to a natural number, the protocol says the argument obeys the `Natural` protocol.

     Usually all the possibilities are covered by a superclass. For example, even though message `size` could be sent to a `List` or an `Array`, you can just draw a single edge to node `Collection/size`. Sometimes you might have more then one outgoing edge per message—for example, if a message could be sent to an `Integer` or a `Fraction`, but not to any `Number`.
  5. If the method is not defined and not inherited from a superclass, the message is *not understood*, and your program is broken.

  Here are some tips about call graphs:

  – *If you have a cycle in the graph, it represents a potential recursion.* Be sure that on every trip through the cycle, some argument or some receiver is getting smaller, or that the algorithm is making progress in some other way. For example, my code for `*` on class `Natural` can sometimes send the `*` message to a natural number, but on every trip through this cycle, the receiver of `*` gets smaller (by a factor of b).

[1] If you encounter a class method, just call the message something like "class method `new`," and proceed as you would otherwise.

  - Have a goal in mind, and ignore messages that are unrelated to the goal. For example, if you are building a call graph to study addition, you probably don't have to include the `max:` message.
  - Loops and conditionals are technically message sends, but I urge you to simplify your call graph by simply assuming that all the code is (eventually) executed.
  - A call graph can help with unit testing: you want to make sure that every solid edge is exercised by some unit test.
  - Like any other analysis technique, call-graph analysis is worth it only when you have a problem. You have an infinite recursion? Or you don't understand how the methods are supposed to work together? Build a call graph. Otherwise, continue to apply your standard design process, and everything will be fine.

A final note: call graphs are new this semester, and not all the TAs are skilled with them yet. You may be learning together.

Assembling documentation for the assignment

The information you need is spread out over multiple sources:

- This handout explains everything you are expected to deliver, and it collects hints and other supplementary information.
- The main textbook explains how Smalltalk works, and it has the protocols of all the classes and the contracts of all the methods—including the protocols and contracts of methods that you are expected to implement.
- The handout "Mastering Multiprecision Arithmetic" explains the algorithms for the arithmetic used in this assignment and the `sml` assignment.
- The Smalltalk bignums handout goes deep into the details of implementing bignums using objects. It focuses on objects and inheritance, not on algorithms.

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. *Receivers, arguments, and messages.* Read the first seven pages of chapter 10, through section 10.1.3. Now examine these expressions from the definition of class `Tikzpicture`, which should be below Figure 10.3 on page 808:

   ```
   (div: w 2)
   (drawOn: shape self)
   (do: shapes [block (shape) (drawOn: shape self)])
   ```

   In each expression, please identify the *receiver*, the *argument*, and the *message*:

   In `(div: w 2)`,

   - The receiver is …
   - The argument is …

- The message is …

In `(drawOn: shape self)`,

- The receiver is …
- The argument is …
- The message is …

In `(do: shapes [block (shape) (drawOn: shape self)])`,

- The receiver is …
- The argument is …
- The message is …

2. *Colons in method names.* Continuing with the analysis of `Tikzpicture`, in both the protocol and the implementation, method `add:` has one colon in the name, method `draw` has no colons in the name, and the method `drawEllipseAt:width:height:` has three colons in the name.

   - What, if anything, does the number of colons have to do with *receivers*?

     Your answer: …

   - What, if anything, does the number of colons have to do with *arguments*?

     Your answer: …

   If you need to, review the presentation in section 10.1.1 on "Objects and Messages," which shows messages sent to shapes.

3. *Class protocols and instance protocols.* Every *message* is part of some *protocol*. As example messages, study the transcript in code chunks 803e and 804, which puts three shapes into a picture and then draws the picture.

   (a) Of the messages used in the transcript, which ones are part of the *class* protocol for `Tikzpicture`, and which are part of the *instance* protocol?

   (b) In general, what do you do with messages in a *class* protocol, and how does that differ from what you do with messages in an *instance* protocol?

4. *Dynamic dispatch, part I: a toy class.* For the mechanisms of message send and dynamic dispatch, read section 10.3.4, which starts on page 820. Using the class definitions in that section, message `m1` is sent to an object of class `C`. What method *definitions* are dispatched to, in what order?

   Please edit this answer to put in the correct methods and classes:

   - Dispatch to method m1 on class ?
   - Dispatch to method ? on class ? …

5. *Dynamic dispatch, part II: number classes.* Study the implementation of class `Number`, which starts around page 881. Now study the implementation of class `Fraction`, which starts around page 885.

   When message `-` (minus) is sent to the `Fraction (/ 1 2)` with argument `Fraction (/ 1 3)`, the computation dispatches message to instance methods of classes Fraction, Number, and SmallInteger, as well as a class method of class Fraction. We are interested in only *some* of those dispatches—ones that meet *both* of these criteria:

- The message is sent from a method defined on class `Fraction` or class `Number`.
- The message is received by an instance of class `Fraction` or class `Number`.

These criteria rule out *class* methods of class `Fraction`, messages sent to `SmallInteger`, and so on.

Starting with message `-` (minus) is sent to an instance of `Fraction`, please identify only the interesting dispatches:

```
Message   Sent from method     Sent to object      Method defined
          defined on class     of class            on class

-           (anywhere)         Fraction            Number
?         Number               ?                   ?
 ... complete the rest of this table ...
```

6. *Dynamic dispatch, part III: messages to `self` and `super`.* Now study the *class* method `new` defined on class `List`, which appears just after page 875. 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 executed, what three messages are sent by the method body, in what order? (If you like, you can also study the message trace shown above, but it may be simpler just to look at the source code.)

   (b) What does each of the three message sends accomplish?

   (c) If we change `new`'s definition so instead of `(new super)` it says `(new self)`, which of the following scenarios best describes how the changed program behaves?

      1) The `new` message will be dispatched to class `List`. The same method will run again, and the computation will not terminate.

      2) The `new` message will be dispatched to a different class, and the reply to the `new` message will leave the sentinel pointing to the wrong value.

      3) Nothing will change; in this example, there's no difference between `(new super)` and `(new self)`.

   Your answer: The best description is scenario number ?

7. *Design of the numeric classes.* Read about coercion in section 10.4.6 on page 839. Look at the last part of the instance protocol for `Number` on page 838. 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`, starting on page 883.

   The role of `asInteger` is …

   The role of `asFraction` is …

   The role of `asFloat` is …

   The role of `coerce:` is …

You are ready to implement mixed arithmetic, with coercions, in exercise 44.

8. *Abstract classes in principle.* In section 10.11.1, which starts on page 929 ("Key words and phrases"), you will find a short definition of "abstract class." What is the *purpose* of an abstract class? Pick one of the responses below.

   (a) To hide the representation of instances so programmers can change internal details without affecting client code

   (b) To define methods that other classes inherit, so that subclasses get useful default methods

   (c) The same as the purpose of a regular class: to define an abstraction

   Your answer: …

9. *Abstract classes in practice: magnitudes and numbers.* Your natural-number class will inherit from abstract class `Magnitude`, and your big-integer code will inherit from `Magnitude` and from `Number`, which is also an abstract class.

   (a) Study the implementation of class `Magnitude`; it is the first dozen lines of code in section 10.7.6, which starts on page 881. List all the methods that are "subclass responsibility":

   ```
   Your answer: ...
   ```

   These are methods that you must implement in both your `Natural` class and your large-integer classes.

   (b) The very next class definition is the definition of abstract class `Number`. Read the first code chunk and again, list all the methods that are "subclass responsibility":

   ```
   Your answer: ...
   ```

   These are the methods that you must implement in your large-integer classes. (Two of them, `+` and `*`, must also be implemented in class `Natural`.)

   You are getting ready to implement large integers.

10. *Double Dispatch.* Read section 10.7.5, which starts on page 880. And read the section "laws for multiple dispatch" in the 7th lesson on program design ("Program Design with Objects"). Now, of the methods on class `Number` listed in the previous question, list each one that needs to know *either* of the following facts about its *argument* (not its receiver):

    - Whether the argument is large or small
    - If the argument is large, whether it is "positive" or "negative"

    For example, `+` is such a method.

    (a) Please list all such methods here:

    Your answer: `+` …

    (b) The methods in part (a) are exactly the ones that require double dispatch. The implementation of each such method sends a message to its *argument*, and the exact message depends on the class of the *receiver*.

    Assume that the receiver is a `LargePositiveInteger`. Please say, for each method in part (a), what message the method implementation sends to the argument.

    Your answer:

Method + sends `addLargePositiveIntegerTo:` to the argument
…

You are ready to implement large integers (exercise 43).

Individual Problem

Working on your own, please work exercise 39(a) on page 948 of *Build, Prove, and Compare*. This exercise is a warmup designed to prepare you for the bignum problems in the pair portion of the assignment.

39(a). *Interfaces as Abstraction Barriers*. Do exercise 39(a) on page 948 of *Build, Prove, and Compare*. Put your solution in file `frac-and-int.smt`. Think about *protocols*, not implementation.

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. If you revise an existing class, you must do so *without* changing the source code. For an example, if you want to revise class `SmallInteger`, you must *redefine* class `SmallInteger` using the idiom on page 950:

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

This idiom enables you to change predefined classes *without* editing the source code of the μSmalltalk interpreter. Using the idiom as needed, you should be able to put your entire solution in file `frac-and-int.smt`.

Hints:

- At minimum, your solution should support addition, subtraction, and multiplication, so include at least one `check-expect` unit test for each of these operations. These tests are run only on your own code, so they do not have to be formatted in any special way.
- 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 class `Integer`, this change doesn't affect class `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 10.4.6, which starts on page 839. Also in that section, read about `Integer` and `Fraction`.
- For the implementation of `Integer`, see page 883. The implementation of class `SmallInteger` is also nearby, but for the time being, you can ignore the details of how it is implemented—almost all the methods are primitive.
- For the implementation of `Fraction`, see page 885. Study the implementation of method `+`, and observe how it relies on the exposure of representation through private methods `num` and `den`.

- Read the section "forms of data, access to representation", which describes three levels of access, in the lesson on "Program Design with Objects".
- If nothing comes to you, try reading about how we get access to multiple representations in the object-oriented way: section 10.7.5, which starts on page 880. You will need to read this section later anyway.

How big is it? 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 work exercise 42 on page 949, exercise 43 on page 949, and exercise 44 on page 950 of *Build, Prove, and Compare*, and exercises **T** and **ADT** below.

Sometimes you want to do computations that require more precision than you have available in a machine word. Full Scheme, Smalltalk, Haskell, Icon, and many other languages provide "bignums," which automatically expand to as much precision as you need. Unlike languages that use abstract data types, Scheme, Smalltalk, and Icon make the transition from machine integers to bignums *transparent*—from the source code, it's not obvious when you're using native machine integers and when you're using bignums. You will build transparent bignums in Smalltalk.

Big picture of the solution

Smalltalk code sends lots of messages back and forth, and those messages are dispatched to lots of methods, which 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.

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 tests, *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. For a `Natural` number, the print operation could be as simple as

```
(method print ()
  (do: (decimal self) [block (digit) (print digit)]))
```

Unit testing bignums in Smalltalk

Arithmetic comparisons should be tested using `check-assert` in the usual way. But other arithmetic operations *can't* easily be tested using `check-expect`, because the parser handles only machine integers. You have two options:

- Use `check-expect` with the `decimal` method.
- Use the new `check-print` form, which is *not documented* in the textbook.

Using `check-expect`

The semantics of `check-expect` are subtle, but the tests work about the way you would hope. You might remember that in μScheme, `check-expect` does not use the built-in `=` primitive—instead, it compares values uses something like the `equal?` function. This comparison enables μScheme's `check-expect` to accept two different S-expressions that have the same structure. In much the same way, μSmalltalk does not use the built-in `=` method—instead, it compares objects using the `similar:` method. This comparison enables μSmalltalk's `check-expect` to accept two different sequences that have the same elements. The details are likely to be important only if you have to debug a failing `check-expect`, but they can be found in section 10.3.5, which starts on page 821.

Using `check-expect` with the `decimal` method looks like this:

```
(check-expect (decimal (x:to: Power 10 60))
              '( 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0))
(check-expect (decimal (x:to: Power 2 64))
              '( 1 8 4 4 6 7 4 4 0 7 3 7 0 9 5 5 1 6 1 6 ))
(check-expect (decimal (negated (x:to: Power 2 64)))
              '( - 1 8 4 4 6 7 4 4 0 7 3 7 0 9 5 5 1 6 1 6 ))
```

Because `check-expect` uses `similar:`, it can accept a *list* of decimal digits returned by `decimal` as *similar to* (but not equal to) the array written using literal quote syntax.

Using `check-print`

Using `check-expect` is familiar, but aside from the obvious awkwardness of writing arrays of decimal digits, it has a more serious flaw: it can't detect bugs in your all-important `print` method. To help find bugs in `print`, I have added a `check-print` unit-test form, which is not yet documented in the book. The `check-print` form takes an expression and a literal result. The literal result must be a single token: either a name or a sequence of digits. Here are some example uses:

```
(check-print (x:to: Power 10 60)
             1000000000000000000000000000000000000000000000000000000000000)
(check-print (x:to: Power 2 64)
              18446744073709551616)
(check-print (negated (x:to: Power 2 64))
             -18446744073709551616)
(check-print (* (new: Natural 256492) (new: Natural 666481))
             170947044652)
```

As soon as you have a `print` method working, I recommend using `check-print`.

Our unit tests

We provide some unit tests for class `Natural` and for bignums. But our tests are not like the unit tests you are used to. We don't hand-craft each test to exercise a particular method in a particular way. Instead, we use a random-number generator to create relatively big tests that exercise many methods at once—especially the all-important `print` method. If your code passes our tests, it should give you some confidence. But if your code fails one of our tests, the failure won't have much diagnostic value. For that reason, our tests are meant to supplement your own unit tests, not to supplant them.

Details of all the problems

42. *Implementing arbitrary-precision natural numbers*. Do exercise 42 on page 949 of *Build, Prove, and Compare*. Implement the protocol defined in Figure 10.17 on page 842. Put your solution in file `bignum.smt`.

The algorithms are the same algorithms you would have used on the `sml` assignment. So please do adapt your code from the `sml` assignment. Or if you prefer, **you may adapt my solutions.** (I provide both an array-based solution and a list-based solution, but the best solution, and the one that is easiest to adapt is the algebraic-datatype solution from the first ML homework.) Whatever code you adapt, be sure to attribute the source!

The choice of a base for natural numbers is yours. But *for full credit, you must choose a base b such that* $b > 10$*,* and small enough that $(b + 1) \cdot (b - 1)$ does not overflow.[2]

What representation should I use? You've got three choices:

- Array-based: Every natural number is represented by an array of digits.
- List-based: Every natural number is represented by a list of digits, where "list" means the mutable Smalltalk `List` abstraction.
- Subclass-based: A natural number's representation depends on its value:
  - The natural number zero is represented by an instance of concrete class `NatZero`, which represents only zero.
  - A natural number of the form "n times base b plus digit d" is represented by an instance of class concrete class `NatNonzero`, which holds instance variables n and d. Objects of class `NatNonzero` represent only *nonzero* natural numbers, so this class has an invariant that n and d are not both zero.

  Both concrete classes should be defined as subclasses of the class `Natural`, which should be an abstract class.

Between the ML warmup solutions and the ML modules solutions, you can compare ML solutions that use different representations. But those comparisons may be misleading: while the array-based solutions are roughly similar, the different way that case analysis is expressed using objects (versus pattern matching) makes the other solutions more different.

The representations offer these tradeoffs:

- The solution based on an array is hardest to get right but easiest to get started. The easy part is that there is no case analysis in the data: every natural number is represented by an array. Hard parts include figuring out array lengths, managing array indices, and making sure not to confuse the *index* of a digit with the digit itself.

[2]Test it.

- The subclass-based solution, with special subclasses for zero and nonzero numbers, is the easiest to get right but the hardest to get started. The easy part is that every method is specialized, so almost all of the case analysis is handled for you automatically, by method dispatch. The hard part is that you have to understand method dispatch.
- A solution based on a *mutable* `List` is possible in principle, but I recommend against it.

In my opinion, the subclass-based solution is clearly superior. (As a bonus, this representation is also the best at helping you learn about programming with objects.) This representation makes it easy for me to understand my own code: I can look at any method, and I am confident that I understand exactly what it is doing. The array and list representations don't give me that confidence.

How must I document my representation? As on the modules assignment, you must document the *abstraction function* and *invariant* for every *concrete* class. (A concrete class is one that is instantiated by sending it the `new` message, or by sending some other message whose method eventually sends `new`.)

- Document the abstraction function by writing, in a comment, an equation

  ```
  A (self) = ...
  ```

  and using instance variables on the right-hand side.
- Document the invariant *either* by writing a comment like

  ```
  I (self) = ...
  ```

  or by defining a private `invariant` method that answers a Boolean.

In each class, place your comments immediately below or to the right of the declaration of the instance variables.

How big is it? Using the hints in the book, I've written two implementations of class `Natural`:

- Using the array representation, my solution is about 120 lines of μSmalltalk code.
- Using the subclass-based representation, my solution is about 150 lines of μSmalltalk code.

I have not written a solution that uses Smalltalk lists.

Related reading:

- There is a detailed implementation guide in the bignums handout. It discusses both array-based and subclass-based representations.
- 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 10.7.5, which starts on page 880.
- In the 7th lesson on program design, read the section on how the design steps are adapted for use with objects. Focus on steps 6, 7, and 8: algebraic laws, case analysis, and results. In the same lesson, you may also wish to revisit the three levels of access to representation. You will need level C, but **there is no need for double dispatch here.**
- Class `Natural` is a subclass of `Magnitude`. Study the `Magnitude` protocol in section 10.4.6. 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` on page 881.[3]

- For the interface to a Smalltalk array, study the `Collection` protocol in section 10.4.5, which starts on page 829. You have access to the protocol in Figure 10.10 on page 832, but you are more likely to use the `KeyedCollection` protocol in Figure 10.11 on page 834, especially `at:` and `at:put:`. Don't overlook the **Arrays** section on pages 879 and 880, 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 10.4.5, especially Figure 10.13 on page 836.
- For abstraction functions and invariants, you may wish to revisit the program-design lesson on abstract data types (lesson 6).

43. *Implementing arbitrary-precision integers*. Do exercise 43 on page 949 of *Build, Prove, and Compare*. Add your solution to file `bignum.smt`, following your solution to exercise 42.

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

The book has starter code for class `LargeInteger`, which you can copy (with acknowledgement) from `/comp/105/build-prove-compare/examples/usmalltalk/large-int.smt`.

How must I document my representation? As on the previous exercise, you must document the *abstraction function* and *invariant* for every *concrete* class. For example if you follow the guidelines and define an *abstract* class `LargeInteger` with two concrete subclasses `LargePositiveInteger` and `LargeNegativeInteger`, you'll need to document only the two concrete subclasses.

How big is it? My solutions for the large-integer classes are 30 lines apiece.

Related reading: This problem is all about dynamic dispatch, including double dispatch.

- Read section 10.7.5, which starts on page 880.
- Read the last section, "Laws for double dispatch," in the 7th lesson on program design.

You'll also have a chance to practice double dispatch in recitation.

Helpful code. I have no wish to torture anyone with the details of signed-integer division. For those of you who might not wish to translate the nested conditionals found in the `divide` function in the μScheme chapter, here are some methods I have defined on my own large-integer classes.

- On class `LargeInteger`:

  ```
  (method div: (n) (sdiv: self n))
  ```

- On class `LargePositiveInteger`:

  ```
  (method sdiv: (anInteger)
    (ifTrue:ifFalse: (strictlyPositive anInteger)
       {(withMagnitude: LargePositiveInteger (sdiv: magnitude anInteger))}
       {(negated (sdiv: (- (- self (new: LargeInteger anInteger))
  ```

[3]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`.

```
                                (new: LargeInteger 1))
                            (negated anInteger)))}))
```

- On class `LargeNegativeInteger`:

  ```
  (method sdiv: (anInteger)
    (ifTrue:ifFalse: (strictlyPositive anInteger)
       {(negated (sdiv: (- (+ (negated self) (new: LargeInteger anInteger))
                           (new: LargeInteger 1))
                        anInteger))}
       {(sdiv: (negated self) (negated anInteger))}))
  ```

I also don't wish to torture anyone with two's-complement representations. The following code on class `LargeInteger` will ensure that the `negated` method defined on `SmallInteger` does not overflow:

```
(class-method new: (anInteger)
   (ifTrue:ifFalse: (negative anInteger)
      {(negated (+ (new: self 1) (new: self (negated (+ anInteger 1)))))}
      {(magnitude: (new LargePositiveInteger) (new: Natural anInteger))}))
```

44. *Modifying `SmallInteger` so operations that overflow roll over to infinite precision.* Do exercise 44 on page 950 of *Build, Prove, and Compare*. Put your solution in a fresh file, `mixnum.smt`. On the first line of file `mixnum.smt`, include your other solutions by writing `(use bignum.smt)`.[4]

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 950:

```
(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.

Related reading: Everything about dispatch and double dispatch still applies, especially the example in the 7th lesson on program design. 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 10.4.3, which starts on page 827.
- Read the description of `at:ifAbsent:` in the keyed-collection protocol in Figure 10.11 on page 834. Now study this expression:

  ```
  (at:ifAbsent: '(0 1 2) 99 {0})
  ```

[4]If there is a bug in your solution to exercise 44, it can break your solutions to the previous exercises. By putting the solution to exercise 44 in its own file, we make it possible to test your other code independently.

This code attemps to access element 99 of the array `( 0 1 2 )`, which is out of bounds because the array only has only 3 elements. When given an index out of bounds, `at:ifAbsent:` sends `value` to the "exception block" `{0}`, which ultimately answers zero.

- Study the implementation of the `at:` method in ⟨*other methods of class* `KeyedCollection` 871d⟩, 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 in exercise 44 on page 950, and study the implementation of `addSmallIntegerTo:` in the code chunk immediately below. That is the technique you must emulate.

T. *Testing Bignums*. In standalone file `bigtests.smt`, 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.

These tests will be run on other people's code, and they need to be structured and formatted as follows:

1. The test must 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 must 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. The code must be included in file `bigtests.smt`.

3. The expected result must be checked using the `check-print` form described above.

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

Here is a complete example containing two tests:

```
; Summary: 10 to the tenth power, linear time, 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-print (run: Test10Power 10) 10000000000)

; Summary: 10 to the 30th power, mixed arithmetic
```

```
(check-print (run: Test10Power 30)
             1000000000000000000000000000000)
```

Here is another complete example:

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

(check-print (value factorial 20) 2432902008176640000)
```

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

ADT. *Collect representations, abstraction functions, and invariants.*

In a new file `adt.txt`, summarize your design work:

- For each *concrete* class you defined to represent natural numbers in exercise 42,
  - List the instance variables.
  - Copy and paste the class's abstraction function and invariant.
- For the natural numbers, explain in one or two sentences *why* you chose the representation that you did. In addition, please tell us what went well and if you have any regrets.
- For each *concrete* class you defined to represent large integers in exercise 43,
  - List the instance variables.
  - Copy and paste the class's abstraction function and invariant.

Two simple sanity checks for multiplication

It's comforting to have an idea that things have started to work. Here are two comforting (but limited) tests. Here, for example, is a version of power:

```
(class Power Object
  []
  (class-method x:to: (x n) [locals half]
    (ifTrue:ifFalse: (= n 0)
      {(coerce: x 1)}
      {(set half (x:to: self x (div: n 2)))
       (set half (* half half))
       (ifTrue: (= (mod: n 2) 1)
         {(set half (* x half))})
       half})))

(check-expect (x:to: Power 2 3) 8)
(check-expect (x:to: Power 3 4) 81)
(check-expect (x:to: Power (/ 1 3) 3) (/ 1 27))
```

And here is code that computes and prints factorials:

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

Depending on your choice of base, this computation may exhaust the "CPU fuel" used to recover from infinite loops—so you may need to test it while running

```
env BPCOPTIONS=nothrottle usmalltalk
```

These tests may be comforting, but they suffer from grave limitations:

- They only ever multiply numbers. They do not add, subtract, negate, or compare numbers.
- They multiply two large numbers. They only ever multiply a large number by a small number, or two small numbers.

These limitations make power and factorial poor tests of correctness.

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. You can generate a string of random digits on the command line by launching the `bash` shell and running this command:

```
for ((i = 0; i < 20; i++)); do echo -n ' ' $((RANDOM % 10)); done; echo
```

You can generate a test command from a list of digits using μScheme:

```
(define nat-test (ns)                        ; alert!  µScheme code
  (letrec ([exp-of (lambda (ns)
                     (if (null? ns)
                         0
                         (list3 '+ (car ns) (list3 '* 10 (exp-of (cdr ns))))))])
    (let* ([left  (lambda () (printu 40))]
           [right (lambda () (printu 41))]
           [space (lambda () (printu 32))]
           [_ (left)]
           [_ (print 'check-print)]
           [_ (space)]
           [_ (print (exp-of (reverse ns)))]
           [_ (space)]
           [_ (app print ns)]
           [_ (right)]
           [_ (printu 10)])
      'Printed!)))
```

For example,

```
-> (nat-test '(1 2 3))
(check-print (+ 3 (* 10 (+ 2 (* 10 (+ 1 (* 10 0)))))) 123)
Printed!
```

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

```
(class Times10 Object
    ()
    (class-method by: (n) (locals p)
        (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
        p))
```

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

You can write, in µSmalltalk instead of μScheme, 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.

Here's an example of using `ghci` to generate numbers for testing:

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude> 256492 * 666481
170947044652
Prelude> 170947044652 * 293847926754
50232434655713663419608
Prelude>
```

Hints and guidelines from past students

Student's experiences is in *italics*; my responds is in the standard font.

- *We found the contracts in the textbook way too late since we were mainly depending on the specification handout and bignums handout. If we had known the contract of each method on each class in the specification, it would have helped immensely.*

  The contracts in the book are crazy useful. The full details for each problem are in the related reading, but for a quick look at some of the most useful information, consult this table:

Class	Pages
`Magnitude`	page 838
`Number`	page 838
`Natural`	page 842
`Integer`	page 840
`SmallInteger`	page 884d
`LargeInteger`	page 949
`Array`[5]	pages 832, 834, and 836

[5]Class `Array` inherits from collections

- *A key issue we ran into that we found out late into the process was when creating the number zero in some arithmetic operations, we were using a constructor that did not enforce the invariant. I think it would be very helpful if the specification put this as a common error or a grave mistake, because most of the logic was correct.*

  Enforcing the invariant is a really good idea. You can define a private method `invariant` and a private method `withInvariant` that looks like this:

  ```
  (method withInvariant ()
    (ifTrue:ifFalse: (invariant self)
       {self}
       {(error: self 'invariant-violation)}))
  ```

- *We weren't using regression testing, so one part would work initially then fail in a later step. It was really helpful in the previous assignments when regression tests were required. Including unit tests as a part of the process and emphasizing the point would also help.*

  You've had two assignments with required regression tests. That's enough for you to decide the value of regression tests.

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 (the "Smalltalk bignums handout") with suggestions about which methods depend on which other methods and in what order to tackle them.

The bignums algorithms are the same as in the ML modules assignment, and in addition to consulting those solutions, you can consult the same references that are recommended in that assignment. In particular, if you insist on the array representation, which is **not** what I recommend, Dave Hanson's book 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 (see also his distribution). 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.

The whole story about division

We do implement division, but not fully. Here's why:

- Without division and modulus, we can't print numbers in decimal. Printing numbers is too useful to ignore.

- *Long* division, in which the divisor of a large number can itself be a large number, is a huge pain in the ass. The algorithm requires trial and error, and it's easy to get wrong.

To enable our code to use division without requiring long division, I have invented messages `sdiv:`, `smod:`, and `sdivmod:with:`. These messages accept only short divisors, and each such divisor must be represented as an object of class `SmallInteger`. And in order to enable these messages to interoperate with Smalltalk's standard integer division, we continue to support the standard messages `div:` and `mod:`. The resulting system design is a little confusing, but here's what we know:

- Method `mod:` is inherited from class `Integer`, and it remains good. As long as division rounds toward minus infinity, `mod:` does the right thing. This means that `mod:` of a large integer by a small integer might return a large integer, even though the result is representable as a small integer. That's OK.
- Methods `sdiv:`, `smod:`, and `sdivmod:with:` are all well specified and well behaved. They don't have to support mixed arithmetic (the argument *must* be a `SmallInteger`, *always*). We just implement each one according to its specification. And because the divisor is always a small integer, there's never a need for double dispatch.[6]
- We provide a limited implementation of `div:` that works only when the divisor is a small integer. On large integers, method `div:` should delegate to `sdiv:`, as shown above. On small integers, the primitive implementation of `div:` can be left alone. And natural numbers don't support `div:`; they only support `sdiv:`.
- Method `reciprocal` is inherited from class `Integer` and also remains good: it allocates a `Fraction` with a large-integer denominator. In a world of mixed arithmetic, arithmetic on such fractions should "just work."

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 a **terrible mistake** to make decisions by interrogating an object about its class—a so-called "run-time type test." Run-time type tests destroy behavioral subtyping. This mistake is most commonly made in two places:

- If you are representing a `Natural` number as a list of digits, you may be tempted to interrogate the representation to ask "are you nil or cons?" This is the functional way of programming, but in Smalltalk, **it is wrong.** You must make the decision by sending a message to an object, and the method that is dispatched to will know whether it is nil or cons.
- If you are mixing arithmetic on large and small integers or on integers and fractions, you may be tempted to interrogate an argument about its class. **This interrogation is wrong.** You must instead figure out how to accomplish your goals by sending messages to the argument—probably including messages from some private protocol.

There is a right way to do case analysis over representations: entirely by sending messages. For an example, study how we calculate the length of a list: we send the `size` message to the list instance. Method `size` is dispatched to class `Collection`, where it is implemented by using a basic iterator: the `do:`

[6]The Fall 2018 printing of the book gets this wrong.

method. If you study the implementation of `do:` on classes `Cons` and `ListSentinel` (which terminates a μSmalltalk list), you'll see the case analysis is done by the method dispatch:

- Sending `do:` to a cons cell iterates over the `car` and `cdr`.
- Sending `do:` to a sentinel does nothing (thereby terminating the iteration).

The idea of "case analysis by sending messages" applies equally well to arithmetic—and the suggestions in the bignums handout are intended to steer you in the right direction. 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 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.

It's not common, but if you rely on the recommended invariant for the subclass-based approach (n and d are not both zero), forgetting to enforce the invariant is a bad mistake.

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.

Speed variations. 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. And measure the time needed to compute the first 50 Catalan numbers.

   The Catalan numbers, which make better test cases than factorial, are defined by these equations (from Wikipedia):

$$C_0 = 1 \qquad C_{n+1} = \sum_{i=0}^{n} C_i \cdot C_{n-i}$$

2. Determine the largest possible base that is still a power of 10. Explain your reasoning. Change your class to use that base internally. *Measure* the time needed to compute the first 50 factorials, and also the time needed to compute the first 50 Catalan numbers.
3. In both cases, measure the additional time required to *print* the numbers you have computed.
4. Specialize your $b = 10$ code so that the `decimal` method works by simply reading off the decimal digits, without any short division. Measure the improvement in printing speed.

5. Finally, try a compromise, like $b = 1000$, which should use another specialized `decimal` method, making *both* arithmetic and decimal conversion reasonably fast. Can this implementation beat the others?

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 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 problem

Submit these files:

- A `README` file containing
  - The names of the people with whom you collaborated
- A file `cqs.small.txt` containing your answers to the reading-comprehension questions
- A file `frac-and-int.smt` showing whatever definitions you used to do exercise 39(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 three unit tests.

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;;;;  Solution to Exercise XXX
(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 exercises you worked (including any extra credit)
  - Narrative and measurements to accompany your extra-credit answers, if any

- A file `bignum.smt` showing your solutions to exercises 42 and 43. This file **must** work with an *unmodified* `usmalltalk` interpreter. Therefore if you use results from exercise 39(a), or any other problem, you will need to duplicate those modifications in `bignum.smt`.
- A file `mixnum.smt` showing your solution to exercise 44. This file should incorporate your other solution by reference, using the line

  ```
  (use bignum.smt)
  ```

  at the beginning. Do *not* duplicate code from `bignum.smt`.
- A file `bigtests.smt` containing your solution to exercise T.
- A file ~~`adt.text`~~ `adt.txt` containing your solution to exercise ADT.

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 see 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