Object-Oriented Programming in Smalltalk

CS 105 Assignment

Due Tuesday, December 13, 2022 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, Objective C, and Swift.

The assignment is divided into two parts.

- On your own, you do a small warmup problem, which acquaints you with pure object-oriented style and with μSmalltalk's large initial basis.
- Possibly with a partner, you implement *bignums* in μSmalltalk. You will complete the following parts:
  - Arithmetic on natural numbers
  - Arithmetic on large signed integers
  - Arithmetic on all combinations of large and small integers
  - Test cases for each of the previous three parts
  - Followup questions to demonstrate what you have learned

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.

Preliminaries

Using the interpreter effectively

If you do not already have it set to run automatically when you login, run this command:

```
use -q comp105
```

You now have access to the μSmalltalk interpreter, `usmalltalk`. To run it interactively with line-editing features, use the command

```
ledit usmalltalk
```

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 sending the messages `printProtocol` and `printLocalProtocol`. These messages, which are shown in Figure 10.10 on page~648, are understood by every uSmalltalk *class*. They provide a quick way to remind yourself what messages an object understands and how the message names are spelled.

To apply the interpreter to a whole file named `solution.smt`, use the command

```
  usmalltalk -q < solution.smt
```

Noting errors in the book

There are two errors in the book that will affect your ability to do the homework. Make note of them:

- In Figure 10.21 on page 676, the protocol gives the wrong specification for the result from `sdiv:`. The result should be a large integer, not a natural number. In total, the specification for `sdiv:` should read as follows:

  > Answer the large integer closest to but not greater than the quotient of the receiver and the argument.

- In Figure 10.22 on page 677, the `new:` message is sent to class `Natural`. The message name should be `fromSmall:`. The correct line reads as follows:

  ```
  {((LargePositiveInteger new) magnitude: (Natural fromSmall: anInteger))}))
  ```

These errors have been corrected in the starter code that we provide.

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online.

Individual Problem

Working on your own, please work exercise~36(a) on page~731 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.

36(a). *Interfaces as Abstraction Barriers*. Do exercise~36(a) on page~731 of *Build, Prove, and Compare*. Put your solution in file `frac-and-int.smt`.

When the problem says "Arrange the `Fraction` and `Integer` classes", the text means that examples like this one should work:

```
((1 / 2) + 3)
```

If you try this with the bare code, you'll see that class `Fraction` sends messages that the argument `3` doesn't understand. The problem can be solved with one of these two approaches:

- You can alter methods on class `Fraction` so they interrogate the argument about its form of data (i.e., its class), then act accordingly. This is the approach that you would use with algebraic data types, when the forms of the argument are limited by a static type system.
- You can alter methods on class `Integer` and add additional methods to class `Integer` until a number like `3` is guaranteed to understand any message that `Fraction` could send it. This is the approach you would use with objects, when you want `Fraction` to work with *any* argument that understands the right protocol, regardless of the argument's class.

Either way, you add or alter methods using the "reflection" interface, in particular the `addSelector:withMethod:` message (Figure 10.10 on page~648). As a (useless) example, I add an `identify` method to the `Integer` class but not to the `Fraction` class:

```
-> (Integer addSelector:withMethod: 'identify
        (compiled-method () ('I-am-an-integer println) self))
<class Integer>
-> (3 identify)
I-am-an-integer
3
-> ((1 / 2) identify)
Run-time error: Fraction does not understand message identify
Method-stack traceback:
  Sent 'identify' in standard input, line 3
```

Message `addSelector:withMethod:` can be used to add new methods or to redefine existing methods.

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.
- You are doing only part (a) of this problem, so the only cases that have to work are cases where the receiver is an instance of class `Fraction`. In particular, we expect that `(3 + (1 / 2))` will *not* work.

Related reading:

- For an overview of the numeric classes and their protocols, read section~10.4.6, which starts on page~658.
- For discussion of how different numbers interoperate, read section~10.7, which starts on page~670. (This section is also important for the pair problems.) The section presents some methods of class `Integer`, but for this problem, the key part is understanding how `Fraction` works.
- Read the section "forms of data, access to representation", which describes three levels of access, in the lesson on "Program Design with Objects".
- The full implementation of class `Integer` is not so important for this problem, but if you want it, you can find it in the Supplement on page 672.

How big is it? You shouldn't need to add or change more than 10 lines of code in total.

Pair problems: Bignum arithmetic

Sometimes you want to do computations that require more precision than you have available in a machine word. Full Scheme, Smalltalk, Haskell, Icon, Python, 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.

By working exercise~37 on page~731, exercise~38 on page~732, and exercise~39 on page~732 of *Build, Prove, and Compare*, you will build transparent bignums in μSmalltalk. You will also work exercises **T** and **F** below. You may work all these exercises with a partner.

Natural numbers

37. *Implementing arbitrary-precision natural numbers*. Do exercise~37 on page~731 of *Build, Prove, and Compare*. Implement the protocol defined in Figure 10.19 on page~662. Put your solution in file `bignum.smt`.

Your design choices:

- *The base of natural numbers.* You must choose a base for natural numbers. *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.[1]
- *Representation.* A natural number is represented by a sequence of digits. But how will that sequence be represented?
  - *An array.* Every natural number is represented by an array of digits. This representation is described in the book on page~679. We provide starter code and a detailed implementation guide.
  - *A list.* Every natural number is represented by a mutable Smalltalk `List` of digits. In ML, this is a good representation. But in Smalltalk, it is not recommended.
  - *Custom classes.* A natural number's representation depends on its value:

[1]Test it.

    - 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.

  This representation is described in the book on pages 679 to 681. We provide starter code and a detailed implementation guide.

  The array representation is **hardest to get right** but easiest to get started.

  The custom-class representation is the **easiest to get right** but the hardest to get started. In my opinion, it is superior.

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 130 lines of μSmalltalk code.
- Using the custom classes, my solution is about 150 lines of μSmalltalk code.

Related reading:

- To learn how to get access to arguments of `+` and `*`, read section~10.7, which starts on page~670.
- 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 **you won't need double dispatch here.**
- If you use arrays, study the interface to a Smalltalk array, which is part of the `Collection` protocol in section~10.4.5, which starts on page~651. Also study the array protocol in Figure 10.13 on page~653, including the class method `withAll:`. Among instance methods, you are more likely to use the `KeyedCollection` protocol in Figure 10.14 on page~656, especially `at:` and `at:put:`. You may also want the class method `new:` that is defined on arrays (page 683).
- To build a Smalltalk list, which you will need for the `decimal` method, look at the `List` protocol in section~10.4.5, especially Figure 10.16 on page~658.

Large (signed) integers

38. *Implementing arbitrary-precision integers*. Do exercise~38 on page~732 of *Build, Prove, and Compare*. Add your solution to file `bignum.smt`, following your solution to exercise~37. We provide starter code (for array natural numbers and subclass natural numbers), which corrects the error in Figure 10.22 on page 677. We also provide a detailed implementation guide.

Because you build large integers on top of `Natural`, you don't have to think about array, list, or subclass representations. 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 30 lines apiece.

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

- Read section~10.7, which starts on page~670.

- 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 the second Smalltalk recitation.

Mixed arithmetic (large and small together)

39. *Modifying* `SmallInteger` *so operations that overflow roll over to infinite precision.* Do exercise~39 on page~732 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)`.[2]

In this problem, you modify class `SmallInteger` and the large-integer classes *without* touching their source code. We provide a detailed implementation guide.

The modifications to `SmallInteger` change the basic arithmetic operations that *the system uses internally*. If your code has bugs, the system will behave erratically. You must restart your interpreter and fix your bugs. Then try again.

How big is it? I added or changed almost 20 `SmallInteger` methods, with an average code size of less than two lines per compiled method. That total includes a number of methods devoted to comparisons. There are also a few additions to large-integer classes.

Related reading:

- Everything about dispatch and double dispatch still applies, especially the example in the 7th lesson on program design.

You also 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~648.
- Read the description of `at:ifAbsent:` in the keyed-collection protocol in Figure 10.14 on page~656. Now study this expression:

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

  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 code chunk 694c, 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 primitives in exercise~39 on page~732, and study the implementation of `addSmallIntegerTo:` in the code chunk immediately below. That is the technique you must emulate.

Testing

T. *Testing Bignums.* In standalone file `bigtests.smt`, you will write 9 tests for bignums:

- 3 tests will test only class `Natural`.

[2]If there is a bug in your solution to exercise~39, it can break your solutions to the previous exercises. By putting the solution to exercise~39 in its own file, we make it possible to test your other code independently.

- 3 tests will test the large-integer classes, which are built on top of class `Natural`.
- 3 tests will test mixed arithmetic involving both small and large integers. (Mixed comparison is optional, so you must not test it—but any combination of addition, subtraction and, multiplication, as large as you can fit under the CPU limit, is fair game.)

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.
4. Code must use only public methods.

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, mixed arithmetic
(check-print (10 raisedToInteger: 10) 10000000000)

; Summary: 10 to the 30th power, mixed arithmetic
(check-print (10 raisedToInteger: 30) 1000000000000000000000000000000)
```

Here is another complete example:

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

(check-print (factorial value: 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.

Final followup

F. After completing the rest of the assignment, please download the text version of the followup questions below and fill in your answers. Many of the questions ask about particular coding scenarios; you are welcome to answer these questions by experimenting with code, by thinking through the scenarios without running any code, or by any combination thereof.

1. While implementing large signed integers, you implement `+` and `*`, and you use methods for `div:` and `mod:` that are provided for you. Why isn't subtraction defined on the large-integer classes? When large integers are subtracted, what code you wrote, if any, is run?

2. C++ has what is called "subclass polymorphism": if a function is has a formal parameter of class C, a caller can pass any instance of class C or any instance of any class C' that *inherits* from C. Class C' is called a *subclass* of C.

   Smalltalk and Ruby have what is called "subtype polymorphism": if a message is expecting an argument that understands protocol P, a caller can pass any object with protocol P', provide that P' understands all the messages in P (and in both protocols, the messages must have with the same meaning). Protocol P' is called a *subtype* of P.

   Answer this question:

   a. After you have modified the predefined classes to complete Exercise 36(a), is the `Integer` protocol a subtype of the `Fraction` protocol? Based on your implementation, justify your answer.

      (If you are submitting as a pair, say *which* individual implementation forms the basis for your answer.)

3. This question illustrates some possibilities that arise when contracts are written without reference to representation. Depending on which representation of natural numbers you implemented, answer **one** of the following two parts:

   a. If you chose the array representation: When message `divBase` is sent to a natural number X, it answers X div b, where b is the base of natural numbers. In the subclass representation, `divBase` usually just answers an instance variable. Explain how you would implement `di-vBase` *efficiently* using the array representation. You may write an explanation in informal English, or you may just write the code.

   b. If you chose the subclass representation: A natural number X can be viewed a sum of digits x_i times powers of b, as in

   $$X = \sum_{i=0}^{n} x_i \cdot b^i, \text{ where } 0 \le x_i < b.$$

   When message `digit:` i is sent to a natural number, it answers x_i. In the array representation, `digit:` is usually implemented by an array lookup. To explain how you would implement `digit:` *efficiently* using the subclass representation, complete the following two method definitions.**Write no English.**

   ```
   (NatZero addSelector:withMethod: digit:
      (compiled-method (i) ...))

   (NatNonzero addSelector:withMethod: digit:
      (compiled-method (i) ...))
   ```

4. This question is about the internals of natural numbers. Depending on which representation of natural numbers you implemented, answer **one** of the following two parts:

   a. If you chose the array representation: Adding private methods `digit:` and `digit:put:` complicates the private protocol for class `Natural`. A simpler protocol might be better. And after the fact, it seems like it would be easy enough to remove these methods by inlining their code. Maybe they should never have been there.

Answer this question: if `digit:` and `digit:put:` had not been suggested, how would your development experience have been different? Do these methods provide enough benefit to justify complicating the protocol?

b. If you chose the subclass representation: Suppose class method `first:rest:` *never* creates an instance of `NatZero`. Suppose instead that an instance of `NatZero` is created *only* by class method `fromSmall:` when it receives 0. Under this supposition, natural-number arithmetic may allocate more objects and send more messages than the code we recommend. But will it still work? If not, what goes wrong? Justify your answer.

5. Suppose the multiplication method on small integers is changed so that it always promotes `self` to a large integer:

```
(method * (anInteger) ((self asLargeInteger) * anInteger))
```

This code seems inefficient, but it has the virtue of being simple. Will it work? If not, explain what goes wrong.

6. If you're trying to design or debug code in a procedural or functional language, you can usually make progress by figuring out what function is being called and inspecting its source code. But with dynamic dispatch, this strategy doesn't work: a single call site often invokes many different methods, and tracing all the paths through the code is almost impossible. (This is why languages like Java and JavaScript require such sophisticated compilers.)

   a. How well did "trust the contract" work for you as a strategy for design and debugging?

   b. Name a message you had to debug where you wanted to see what code would run when the message was sent, but it was difficult. (If there was no such message, name a message where you wanted to see code but *didn't* need to debug. If you never wanted to see code, then explain how you handled `multiplyBySmallInteger:` without wanting to see code.)

   c. Was the message associated with a contract? If so, what was the contract? Was the contract testable?

   d. In a functional language, ideal unit tests exercise all combinations of forms of input data. In an object-oriented language, what analogous thing should unit tests ideally exercise?

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.

The bignums algorithms are the same as in the first ML assignment and the ML modules assignment. You are welcome to adapt your solutions or mine.

How to write recursions that terminate

For most beginning Smalltalk programmers, infinite recursion is a significant problem. In Smalltalk, recursions are often mutual and hard to trace. To eliminate "recursion too deep" errors, consider the guidelines below.

- *Diagnosis.* If you get "recursion too deep" in a unit test, you won't know for sure where the problem is. Your first step is to get a stack trace by running the faulty code interactively, *outside* a unit test:

  ```
  ((Natural fromSmall: 7) * (Natural fromSmall: 99))
  ```

  You may discover that the infinite recursion you thought was in `*` is actually in `plus:carry:`.

- *Natural numbers (exercise~37).* In methods like `plus:carry:` and `*`, when you send a recursive message, be sure that *the number of digits in the* ***receiver*** *of the new message is* ***smaller*** *than the number of digits in an existing number.* That existing number may be either the receiver or the argument of the method that sends the recursive message.

  In multiplication, it is common to send the `*` message to a natural number that is *not* guaranteed to have fewer digits. Infinite recursion ensues.

- *Large integers (exercise~38).* At each recursion the number of objects of unknown class has to get smaller. For example, when an object receives the `*` message, the class of the receiver is known but the class of the argument is unknown. When an object receives the `multiplyByLargePositiveInteger:` message, the classes of *both* the receiver and the argument are known. If `multiplyByLargePositiveInteger:` then sends `*` to a large integer, the number of unknowns isn't getting smaller, and in fact that's an infinite loop.

  Sending messages *from* large integers *to* objects of class `Natural` shouldn't cause any loops, because none of the `Natural` methods sends messages to large integers.

- *Mixed arithmetic (exercise~39).* In mixed arithmetic, every potentially recursive operation should decrease one of the following two quantities:

  - The number of small integers not yet promoted
  - The number of objects whose class is unknown (provided the number of small integers not yet promoted stays the same)

  For example, if a small integer receives `*` and sends `multiplyBySmallInteger:`, then it decreases the number of objects of unknown class. And if the `multiplyBySmallInteger:` method defined on class `LargeInteger` *promotes* its argument, then it has decreased the number of small integers not yet promoted, so it is OK for it to use `*` again, even though `*` increases the number of objects whose class is unknown.[3]

  Mixed arithmetic *also* must ensure that in any recursion, the number of digits in the operands is decreasing. That job is mostly done for you by class `Natural`, but there is still one potential infinite recursion that is truly insidious. That recursion arises if you try to promote every operation to be done on large integers. For example,

  - Message `*` is sent to a small integer, which promotes.
  - Message `*` is then sent to large integers, which delegates to `Natural`.
  - Method `*` on class `Natural` then breaks down its operands by digits, and it begins by multiplying two least-significant digits $x_0 \cdot y_0$.

[3]The object that receives `*` must always treat its argument as an object of unknown class.

    - The multiplication $x_0 \cdot y_0$ sends * to a small integer, which promotes…

  This recursion is the easiest one to avoid: *doing an operation on two small integers must always invoke a* `primitive` *expression.* To see how this is done, look at the definitions of the methods on class `SmallInteger` (in `predefined.smt`). And to see an example of a version that detects overflow, look at page~732.

Avoid common mistakes

Below you will find some common mistakes to avoid.

Simple coding mistakes

There are two common mistakes that lead to `Name not found` errors:

- In a continuation argument, such as to `ifTrue:ifFalse:` or `sdivmod:with:`, you forgot to put round brackets inside the curly brackets.
- While editing or uploading a file, you inadvertently introduced a Unicode character into your code. These characters can be found on the Unix command line by using `grep`:

  ```
  env LC_ALL=C grep -n '[^[:print:]]' bignum.smt mixnum.smt
  ```

Some of these mistakes can be detect with `lint-usmalltalk`. You might get lucky.

Common mental mistakes

It is a common mistake to **ignore the design process**. Don't be like the students who said,

> *We followed the design process for assignments where we were to design individual functions but when the specification told us what to do stepwise, like in the Naturals homework, we didn't.*

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. When in doubt, "don't ask; tell."
- 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:`

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 step-by-step guides 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 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`".

In exercise T, it's surprisingly common to fail to tag the test summary with the prefix `Summary:`, or to forget it altogether.

In exercise~39, it's a common mistake to try to implement `(self - anArgument)` by evaluating `(anArgument - self)`—or more precisely, by evaluating `((anArgument asLargeInteger) - self)`. This recursion terminates, but it produces an answer with the wrong sign. If you get "Array index out of bounds" errors, look for this mistake.

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.

Four diagnostic techniques

To help you diagnose problems in your code, we recommend four diagnostic techniques: stack tracing, static spell checking, dynamic message tracing, and static call-graph analysis.

Stack tracing

In Smalltalk programs, many faults manifest as checked run-time errors, like "name not found" or "message not understood." When the interpreter detects a run-time error, it will show you a *stack trace* of every method that was active when the error occurred. But you get the trace only if the expression is evaluated *outside* of any unit test. If a fault occurs in a unit test, you won't automatically get a stack trace. To get one, you must copy the offending expression and paste it into the interactive read-eval-print loop.

A stack trace shows every active message to which the interpreter is waiting for a reply. For example, if I look up a key in an empty dictionary, the error isn't discovered until half a dozen methods are active:

```
-> (val empty (Dictionary new))
Dictionary( )
-> (empty at: 'cs105)
Run-time error: key-not-found
Method-stack traceback:
  In predefined classes, line 425, sent `error:` to an object of class Dictionary
  In predefined classes, line 427, sent `value` to an object of class Block
  In predefined classes, line 434, sent `value` to an object of class Block
  In predefined classes, line 427, sent `associationAt:ifAbsent:` to an object of class Dictionary
  In predefined classes, line 425, sent `at:ifAbsent:` to an object of class Dictionary
  In standard input, line 3, sent `at:` to an object of class Dictionary
```

```
->
```

The most recent pending message appears at the top. Each line shows where in the source code a message was sent from, the name of the message, and the class of the receiver.[4] Use the trace to find exactly what was running when things went wrong.

Tools that analyze your code for potential faults

Smalltalk has no type system. But some properties of of your code can still be checked statically:

- A "wrong number of arguments" check is built into the language. A symbolic message like `+` or `!=` expects exactly one argument (not counting 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—not counting 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 spell 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
  ```

Message tracing

Almost all run-time issues can be resolved with a stack trace. But if you are getting wrong answers with no errors, then as a last resort, you can trace every message send and reply involved in evaluating an expression. Just put the expression in a block and send `messageTrace` to the block. Here is an example message trace of a block that sends message `new` to class `List`, which is the subject of one of the reading-comprehension questions:

```
-> ({(List new)} messageTrace)
standard input, line 5: Sending message (value) to an object of class Block
  standard input, line 11: Sending message (new) to class List
   predefined classes, line 593: Sending message (new) to class SequenceableCollection
    predefined classes, line 593: (<class List> new) = <List>
    predefined classes, line 593: Sending message (new) to class ListSentinel
      predefined classes, line 581: Sending message (new) to class Cons
      predefined classes, line 581: (<class ListSentinel> new) = <ListSentinel>
    predefined classes, line 582: Sending message (pred: <ListSentinel>) to an object of class ListSentinel
    predefined classes, line 582: (<ListSentinel> pred: <ListSentinel>) = <ListSentinel>
    predefined classes, line 583: Sending message (cdr: <ListSentinel>) to an object of class ListSentinel
    predefined classes, line 583: (<ListSentinel> cdr: <ListSentinel>) = <ListSentinel>
    predefined classes, line 593: (<class ListSentinel> new) = <ListSentinel>
   predefined classes, line 593: Sending message (sentinel: <ListSentinel>) to an object of class List
    predefined classes, line 593: (<List> sentinel: <ListSentinel>) = <List>
```

[4]Unless the message was sent to `super`. In that case, the stack trace shows the class where the method search started.

```
  standard input, line 11: (<class List> new) = <List>
standard input, line 5: (<Block> value) = <List>
List( )
```

An analysis you can do by hand

When all else fails, there is a static analysis you can do by hand 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:[5]

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).
- 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.

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

- 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: not all of our TAs have used call graphs before. You may be learning together.

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?
6. For subtraction, Implementing `sub:withDifference:ifNegative:` with a private method `minus:borrow:` requires you to compare the receiver and argument since `minus:borrow:` should only be called to compute non-negative differences. Doing both the comparison and the subtraction redundantly traverses the receiver and argument representations. Convert the `minus:borrow:` algorithm to a CPS style in a method called `minus:borrow:withDifference:ifNegative:` that takes a success and failure continuation so that `sub:withDifference:ifNegative:` does not need to do the comparison. Measure the time difference between subtractions using these two implementations for big number subtractions.

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. For the algorithm, see Per Brinch Hansen, *Multiple-length Division Revisited: a Tour of the Minefield. Software—Practice & Experience*, 24(6): 579-601.

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.

Representation comparison. For the truly dedicated, implement *both* the array and subclass representations, then compare their speed:

- Do experiments that enable you to predict roughly how much slower the subclass representation is.
- Explore the computation so you can explain *why* the subclass representation is slower. You explanation should be quantitative; count messages.

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 `frac-and-int.smt` showing whatever definitions you used to do exercise~36(a). It probably sends message `addSelector:withMethod:` to one or more of these classes: `Fraction`, `Integer`, and `SmallInteger`. And it most definitely includes at least three unit tests.

Please identify your solution 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
  - 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 37 and 38. This file **must** work with an *unmodified* `usmalltalk` interpreter. Therefore if you use results from exercise~36(a), or any other problem, you will need to duplicate those modifications in `bignum.smt`.

  Also, the file must load **without any warnings**.

  If you wish, you may include unit tests in this file, provided the result meets these requirements:

  - Every included unit test must pass.
  - The entire file must load in under 2 CPU seconds.

  Longer-running unit tests can go into file `longtests.smt`.

- A file `mixnum.smt` showing your solution to exercise~39. 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`.

  This file must also load **without any warnings**.

  If you wish, you may include unit tests in this file, provided the result meets these requirements:

  - Every included unit test must pass.
  - The entire file must load in under 2 CPU seconds.

  Longer-running unit tests can go into file `longtests.smt`.

- Optionally, a file `longtests.smt` containing as many unit tests as you wish. There is no limit on the amount of time these tests may take.

- A file `bigtests.smt` containing your solution to exercise T.

- A file `small-followup.md` containing your (joint) answers to exercise F.

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 an appropriate method of its class. • Unit tests are indented by	• 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