CS442
Module 8: Object-Oriented Programming

University of Waterloo

Winter 2025

> "Object-oriented design is the roman numerals of computing."
>
> — Rob Pike

1 Object-Oriented Programming

In this module, we discuss *Object-Oriented Programming* (OOP), a paradigm that has enjoyed considerable popularity in recent decades. Proponents of OOP cite its potential for information hiding and code reuse; opponents argue that the hierarchical type systems imposed by object-oriented languages are not always an accurate reflection of reality, and can lead to compromised and unintuitive type hierarchies.

Objects were first introduced in 1967, as part of the programming language Simula, a descendent of Algol[1]. While Simula had many of the features we now attribute to OOP, they came almost by accident from its goal of simulation (hence the name). Later languages, in particular our exemplar, Smalltalk, are responsible for expanding on and refining them into modern OOP.

However, the widespread adoption of OOP into the programming mainstream did not happen until decades later. OOP grew from fairly niche to major importance in the 1990s. Currently, OOP is one of the most popular (probably the most popular) paradigms among programmers; most widely-used modern programming languages have some kind of support for OOP. However, there are relatively few languages that conform strictly to the object-oriented mentality and may thus be legitimately considered "purely" object-oriented. Smalltalk is one such language, but it's difficult even to name a second, setting aside research languages and languages with no modern maintenance. Instead, OOP acts more as a "meta-paradigm" that may be combined with other paradigms. For example, Ada95 and C++ are fundamentally structured, procedural languages, with support for objects in addition to that procedural core. Java conforms more strictly to the object-oriented mentality than do C++ and Ada (as Java forces programs to be organized into classes), but the language in which methods are written in Java remains fundamentally structured programming. On the other hand, languages like CLOS and OCaml add object-orientation to functional languages.

In Module 1, when we introduced Smalltalk, it was described as being "so object oriented that it's barely a procedural imperative language". It is the complete absence of traditional procedures, the fact that even simple numbers are objects, and the encapsulation of conditions and loops—structured programming—into objects that makes Smalltalk purely object-oriented. Mostly-OOP languages like Java eschew this level of OOP purity in favor of familiarity and predictability.

1.1 What OOP Is(n't)

Because of its current widespread popularity, object-oriented programming is a particularly difficult paradigm to study in the abstract. Part of the reason for the difficulty is that there is no widespread consensus among programmers and language designers about what the defining features of an object-oriented language should be.

[1] Algol was, in turn, was a major early procedural language.

Here we will present the most common language features possessed by object-oriented languages. When we discuss semantics, we will discuss it in the context of Smalltalk, with occasional sidebars to discuss how other languages differ.

At the very least, OOP has *objects*: the encapsulation of data and behavior into a single abstraction, which can be viewed equivalently as records with associated code, or code with associated records. Some other language features commonly associated with object-oriented programming are outlined below:

- *Reference Typing*: variables and fields can only hold references to objects. There is no primitive type that is not (at least apparently) a reference, and there are no bare records;
- *Data Abstraction*: objects often provide facilities by which we may separate interface from implementation, often hiding parts of the implementation behind abstraction barriers;
- *Inclusion Polymorphism*: object types tend to be arranged in hierarchies by a subtyping relationship that allows some types to be used in place of others;
- *Inheritance*: objects often share some of the details of their implementation with other objects, but may override particular methods freely (note: in C++, only virtual methods are truly OO);
- *Dynamic Dispatch*: the actual code associated with a particular method invocation may not be possible to determine statically. Dynamic dispatch is a form of dynamic binding, which we briefly mentioned in Module 2.

Most object-oriented languages possess at least some of the above characteristics.

2 Exemplar: Smalltalk

You've already learned and used Smalltalk in this course, so there's no need to introduce its syntax here. Instead, we'll discuss Smalltalk's place in language history.

Smalltalk was developed at Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, and many others. You may have heard of Xerox PARC; it's a research group famous for inventing everything before anyone else and then failing to monetize any of it. Among their various inventions are:

- the graphical user interface, later monetized by Apple and Microsoft;
- What-You-See-Is-What-You-Get (WYSIWYG) editing, later monetized by numerous corporations;
- "fully-fledged" object-oriented programming (in Smalltalk), later monetized by Sun and later still by numerous corporations;
- prototype-based object orientation (which we will briefly look at in this module), later monetized by Netscape as JavaScript.

Xerox is known for printers.

In fact, the first and third items in that list are one and the same: Smalltalk! The graphical user interface that Xerox PARC was famous for inventing *was* Smalltalk. Indeed, a course on the history of human-computer interaction (HCI) would probably discuss Smalltalk as a major leap forward in HCI, with only a brief mention of the fact that it's also a programming language. It's nearly impossible to separate the two concepts, because of the experience of programming in Smalltalk: there is no such thing as a Smalltalk file[2], and to write Smalltalk code, one interacts with a Smalltalk environment in which they can graphically create and define new classes. The Smalltalk language and implementation were described thoroughly in the so-called "blue book", *Smalltalk-80: The Language and its Implementation* **?**. The Smalltalk programming environment was described in the so-called "red book",

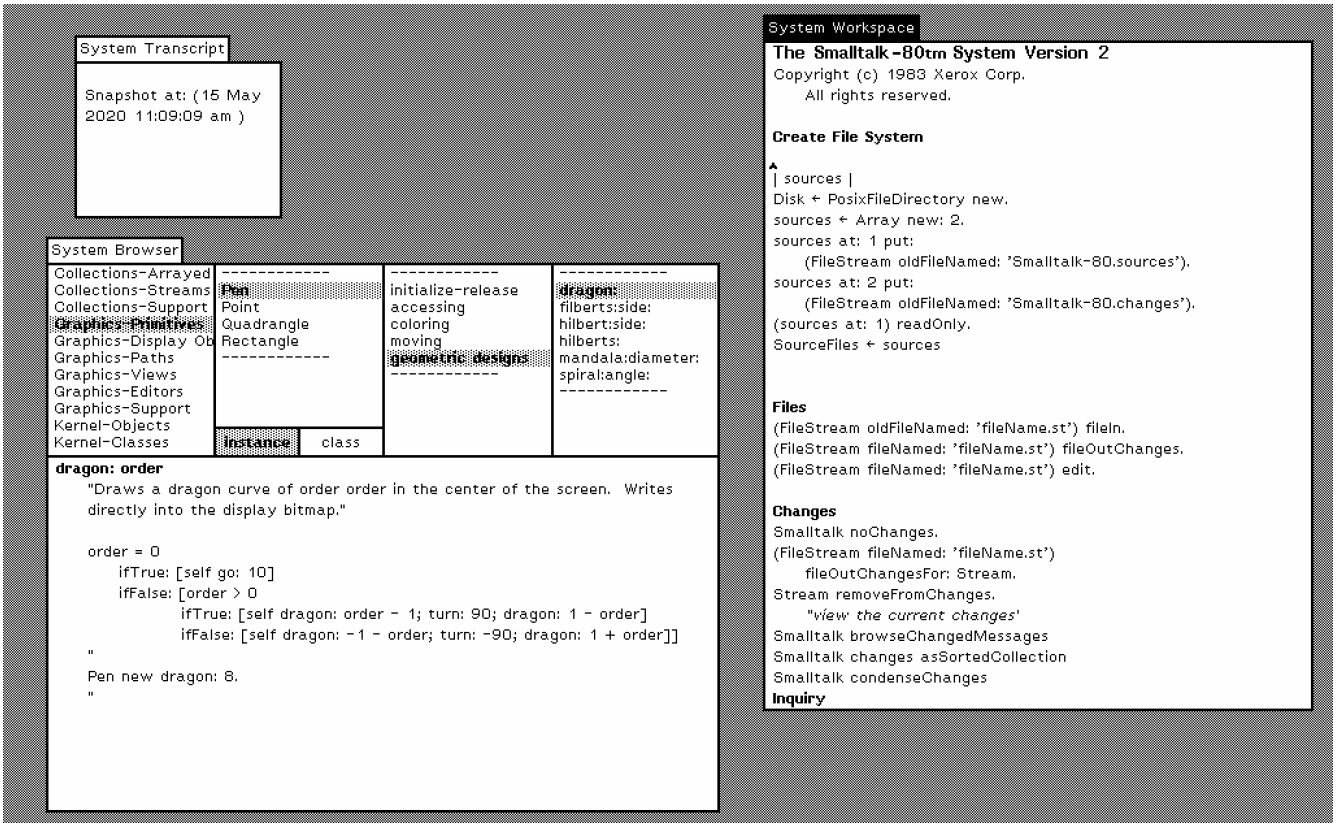

Figure 1: The Smalltalk environment, from a faithful recreation created by Dan Banay at https://github.com/dbanay/Smalltalk. The system browser is used to create and modify classes graphically, so there is no syntax for classes themselves. Shown is a method of the Pen class.

Smalltalk-80: The Interactive Programming Environment **?**. An example of a Smalltalk environment is shown in Figure **??**.

Smalltalk's design throws out the idea of a main function or starting point, and instead opts for the program and interface to be a uniform medley of objects. This is useful for keeping Smalltalk quite pure in its OOP design: in Smalltalk, everything is an object. But, we need many objects to already exist to even perform basic functions (think of Smalltalk's `true` and `false`), so it's hard to rectify this design with a "starting point". Instead, Smalltalk software is distributed as images, which are essentially a frozen state of one of these object medleys; one loads an image and is then experiencing the same environment as that in which the programmer wrote their software.

Ultimately, much of that design, while central to Smalltalk's philosophy, isn't relevant to our interest in object-oriented programming. GNU Smalltalk gets around it by having a special file syntax and semantics. We'll get around it by borrowing a concept from Java: a main class with a main method. But, we're getting ahead of ourselves; let's focus on the basic elements of OOP.

Smalltalk is untyped, so when discussing types, we'll use an extended syntax borrowed from *Strongtalk* **?**, a typed[3] variant of Smalltalk. In Strongtalk, the types of fields, variables, arguments to methods, and returns from methods are annotated with explicit types, like so:

```
1 hypotenuseWithSide: x <Number> andSide: y <Number> ^<Number> [
2     | x2 <Number> y2 <Number> |
3     x2 := x*x.
4     y2 := y*y.
5     ^(x2 + y2) sqrt
6 ]
```

The `<Number>` next to each argument indicates that that argument is of type `Number`, and the `^<Number>` at the end indicates that this method returns a `Number` as well. Like in the Simply Typed λ-calculus, the return type can be discovered from the return statements in the method, so it doesn't need to be explicitly specified. But, as methods may have multiple return statements in Smalltalk, we'll specify it explicitly.

3 Classes

Most—but not all—object-oriented languages use *classes* to describe objects, and can be described as class-based languages. A class is a description of a set of objects with identical behavior and identical form, but (presumably) different internal state. This definition is similar to the definition of a type, and indeed, classes form the types of an object-oriented type system. For now, let's just focus on semantics.

In a purely object-oriented, class-based language, the global scope contains only classes, and all imperative code must be boxed into classes. Even in GNU Smalltalk, which violates OOP purity by allowing behavior outside of classes, the global scope contains only classes; GNU Smalltalk also has a *file scope*, and an individual file may have other elements in its scope, but if you load multiple files, they can only see each others' classes.

A class contains *fields* and *methods*. The fields are conceptually the same as fields of a record from imperative languages, and in that aspect, classes can be considered to be an extension of records: every object of a given class has values for each field declared in the class. Purely object-oriented languages such as Smalltalk provide no syntax for accessing the fields of another object, so that the particular fields of an object are an implementation detail, but most object-oriented languages provide a way to break this encapsulation. Public fields can always be rewritten in terms of getter and setter methods, so the distinction is largely in intent.

Methods are conceptually similar to procedures, but they are associated with classes, and a given method is called *on* an object of the class, so that a given method will always be called with an appropriate object. There is a now-simmering war between two camps on how to describe methods: Smalltalk proponents usually describe *messages* which are matched to methods, and the methods are then *invoked*. In Java, C++, and most other object-oriented languages, methods are simply "called", like procedures. There is no practical difference between them, and

[2]GNU Smalltalk is an outlier and exception to all the rules. Any mentions of "Smalltalk" without "GNU" as a qualifier are referring to traditional Smalltalk, not GNU Smalltalk.

[3]Technically, *optionally typed*: you may specify types if you desire in Strongtalk, but are not required to, and there is no type inference, so code without types behaves as in Smalltalk.

we only fight over this question of language because programming language people are, after all, language people.

The object that a method is called on is the *receiver* of the message if you're using Smalltalk terms, and the *target* of the call otherwise. It is sometimes conceptually convenient to describe the target as a "hidden parameter", and that is indeed how implementations of object-oriented languages work, but taking this concept too far will make typing of class-based languages inconsistent, so be careful of it. Within a method, the target object can be accessed, typically with the name `self` or `this`, depending on the language.

We'll use GNU Smalltalk's syntax for classes. In GNU Smalltalk, we declare a class—we'll ignore for the moment the subclassing that this introduces—like so:

```
Rectangle subclass: Square [
  " content of the Square class... "
]
```

This introduces the name `Square` into the global scope, referencing the class. The class can be used to create objects, with, in this case, `Square new`. There are as many syntaxes for creating objects of a given class as there are object-oriented programming languages, so we won't explore any others.

Let's create a syntax for class declarations. For the time being, that's the only syntax we'll need:

$$
\begin{aligned}
\langle prog \rangle &::= \epsilon \\
&\mid \langle classdecl \rangle \, \langle prog \rangle \\
\langle classdecl \rangle &::= \langle var \rangle \; subclass : \langle var \rangle \, [\langle fieldsdecl \rangle \, \langle methodslist \rangle] \\
\langle fieldsdecl \rangle &::= \epsilon \\
&\mid \text{“|”} \, \langle varlist \rangle \, \text{“|”} \\
\langle varlist \rangle &::= \epsilon \\
&\mid \langle var \rangle \, \langle varlist \rangle \\
\langle methodslist \rangle &::= \cdots
\end{aligned}
$$

Thus, a program is a list of class declarations, and a class declaration has four parts: a superclass, a name, field declarations, and method declarations. We haven't defined the syntax for methods, but as you've already seen Smalltalk, we'll use its syntax in examples.

Note that in the file itself, nothing *does* anything. We've encountered this phenomenon before: in most typed functional languages, you can only write declarations into a file, not expressions. In a purely object-oriented language, you can only write class declarations into a file, not statements.

> **Aside:** In this sense, purely object-oriented languages are declarative: declarations are primary, not behavior. Indeed, the same thing tends to happen with typed procedural languages. For historical reasons, the "declarative" paradigm is considered opposite to imperative, and so has less to do with declarations than referential transparency.

When a class is declared, it is declared a *subclass* of some existing class, which is in turn its *superclass*. In our simple `Square` example, `Square` was declared as a subclass of the `Rectangle` class. This means that it *inherits* all of the fields and methods from `Rectangle`. At its most basic level, this inheritance is very simple: if `Rectangle` has a field named `width`, then `Square` also has a field named `width`; if `Rectangle` has a method named `area`, then `Square` has an identical method named `area`. But, `Square` may be defined with additional fields and methods as well as those defined by `Rectangle`.

Finally, `Square` may *re*-define methods that were defined in `Rectangle`: instead of inheriting the behavior of a method, it may *override* the behavior of a method. In such an overridden method, you may call the superclass's original implementation, or any other method of the superclass, typically with a special syntax that looks like calling the method on the object `super`. For instance, consider this partial implementation of `Rectangle` and `Square`:

```
Object subclass: Rectangle [
    | width height |
    " ... constructor, etc... "

    setWidth: v [
        width := v.
    ]

    setHeight: v [
        height := v.
    ]

    setWidth: w setHeight: h [
        self setWidth: w.
        self setHeight: h.
    ]

    area [
        ^width * height
    ]
]

Rectangle subclass: Square [
    setWidth: v [
        super setWidth: v.
        super setHeight: v.
    ]

    setHeight: v [
        self setWidth: v.
    ]
]
```

By overriding the `setWidth:` and `setHeight:` methods, a `Square` guarantees that it will always be, well, square: the `setWidth:` method calls `Rectangle`'s `setWidth:` and `setHeight:` methods on the same value, and the `setHeight:` method calls `Square`'s `setWidth:` method. There's no need to override the `setWidth:setHeight:` method, since it calls `setWidth:` and `setHeight:`, but it's important to note the behavior of `self setWidth:` and `self setHeight:` within the `setWidth:setHeight:` method: if `self` is a `Square`, then this will call `Square`'s method even though it's actually part of the implementation of `Rectangle`! Any method which is not overridden is inherited, so regardless, the subclass is guaranteed to have at least all the same methods as the superclass. Some languages require explicitly specifying when a method is intended to override a superclass method.

When a method is dispatched, it is the class of the actual object at run-time that determines which method implementation (the original or an override) will be called, *never* the static type. Method dispatch intentionally obscures what method implementation is actually called, as subclasses—including subclasses the calling code is unaware of—may override methods of superclasses. In this way, code that uses objects is separated from the code that implements the objects, allowing modular substitution of methods—at least, ideally. Note that in C++, methods are non-virtual by default, which is quite contrary to the spirit of object-oriented programming; methods are a concept that exists exactly for this virtualization, so hobbling that renders C++ non-OO by default.

> **Aside:** I will repeatedly pick on C++ when mentioning many features. This isn't because of particular animosity to C++; it's just expected that many programmers have learned OO with C++, and unfortunately, if they have, then they've learned a very peculiar derivative of OO. It's necessary to mention the many ways that C++ is not OO (or atypically OO) so that programmers accustomed to it know when to discard this prior knowledge.

Since classes define objects with the same fields and methods, the implication of this inheritance is that an object of the class `Square` *is* an object of the class `Rectangle`: it has at least all the same fields and methods, so anything you can do with a `Rectangle`, you can do with a `Square`. But it may have more than `Rectangle` has. So, all `Square`s are `Rectangle`s, but not all `Rectangle`s are `Square`s. This requirement also mandates the use of references to objects: if `Square` added additional fields, then it would need more space, but a local variable of type `Rectangle` must also accept a `Square`, so instead the local variable simply stores a reference, which is always the same size. Once again, C++'s violation of this makes C++ non-OO by default; you must use pointers or references to objects

in C++ for subclasses and their superclasses to be truly interchangeable.

4 Semantics

At the basic level, the semantics of an object-oriented language follow naturally from the semantics of an imperative language. However, as basic mathematical operators are—or at least, can be—methods, we don't need most of the weight from even the Simple Imperative Language. Instead, we will define a simple(ish) semantics, with some parts based *very* loosely on *Featherweight Java* **?**, for an object-oriented language in the style of Smalltalk.

We need a few exceptions to Smalltalk to make the language tractable to formally define its semantics. First, we will make field access syntactically distinct from local variable access, by imagining that fields are accessed with an arrow, e.g. `self->width`, as in C++. Since fields and local variables are both explicitly declared, it would be trivial to modify existing Smalltalk code in this way. Second, because Smalltalk has several different syntaxes for methods, we require that all user-defined methods be of the colon-separated form, such as `setWidth:setHeight:`. We will define two zero-argument methods as built-in methods, and this will assure that they are separate and distinct from user methods, as well as just simplifying the rules for calling methods. Third, we require that every method have exactly one return statement, and that that return statement be the last statement in the method. This is because return's ability to break out of a method early makes the semantics much harder to define. Finally, we remove expression syntax entirely by requiring that every statement be of one of a limited set of forms:

- `x := M`, where `x` is a variable name, and `M` is a method call. Both the target and the arguments to the method call must be variable names.
- `x := [ ... ]`, i.e., an assignment of a block to a local variable.
- `x := y`, i.e., an assignment of a variable to another variable.
- `x := self->y`, i.e., the transfer of a field of `self` to a local variable.
- `self->y := x`, i.e., the transfer of a local variable to a field of `self`.
- `^x`, i.e., a return statement returning from a local variable.
- `x`, just a variable name.

For instance, we would rewrite

```
x := r setWidth: (s area).
```

as

```
x1 := s area.
x := r setWidth: x1.
```

Since even mathematical operators are methods in Smalltalk, any expression can be broken up into individual steps in this way.

The store, σ, will contain global class declarations and, like in our semantics for the Simple Imperative Language, "freshened" local variables. We also need a heap, Σ, to store our objects, and labels to reference them[4]. There is no syntax for an object value in Smalltalk, so we'll have to define one: an object is defined by its class and the values of all of its fields. So, we will give its syntax in the heap like so:

$$\begin{aligned} \langle object \rangle &::= \langle var \rangle \, [\langle fieldvaluelist \rangle] \\ \langle fieldvaluelist \rangle &::= \epsilon \\ &\quad \mid \langle var \rangle \text{“:=”} \langle value \rangle \, . \, \langle fieldvaluelist \rangle \end{aligned}$$

The $\langle var \rangle$ in the definition of $\langle object \rangle$ is the object's class. Each field is written $\langle var \rangle := \langle value \rangle$, where the first $\langle var \rangle$ is the name of the field, and $\langle value \rangle$ is the field's value. Technically, Smalltalk allows multiple fields to have

[4]Some semantics manage to combine σ and Σ by making objects immutable (reference typing is indistinguishable from copying if nothing can be modified anyway), but mutability is sufficiently important that we define then separately.

the same name if they're in different classes, but we'll see in Section **??** that this can easily be written away, so we'll stick to just field names. The values in Σ will all be $\langle object \rangle$s, and the values in σ, along with the globally defined classes, will be labels for objects in Σ.

In fact, we need one more type in σ: blocks. Smalltalk can't operate at all without blocks, so we'll need them to make sense of anything. Since blocks evaluate to a value, we will restrict them similarly to methods: the last statement in the block (which is equivalent to the return statement) must just be a variable name, which will be the value that the block evaluates to.

Like with Haskell, our global scope contains only declarations, so we need a *resolve* function to populate σ with all of the defined classes.

Exercise 1. Define *resolve* for Smalltalk, given the restriction that files contain only class declarations.

Also like with Haskell, since our files contain only class declarations, we need a starting point. GNU Smalltalk's answer to this is to allow files to contain statements, but that breaks object-orientation purity; other Smalltalk implementations' answer to this is to be an environment rather than a program, but that's not very useful for defining semantics. We will borrow a concept from Java by having a main class, which we will distinguish simply by requiring to to be named `Main`, and requiring that it have a main method, which we will require to be named `main:`. Thus, we start our execution with a σ already populated with classes, and with the following standard statements to "bootstrap": `x := Main new. x := x main: x.`. Note that the argument to `main:` is useless, and is only required to fit the restrictive syntax we defined above.

Let the metavariables L, Q, O, M, $v - w$, $x - z$, and ℓ range over statement lists, statements, objects, method declarations, values, variable names, and labels, respectively.

There are only a small number of possible statements, so we simply need to define a semantics for each of them. Unfortunately, a few method calls require special implementations. First, the built-in method call for creating a new object, `new`. Because of inheritance, `new` needs a way of gathering all of the fields in all of the superclasses of the named class. We will define this as $fields(\sigma, x)$, which returns a list field names for all the fields in x and all of its superclasses. As a practical matter, `new` always has a special implementation in the Smalltalk language implementation, and cannot be implemented directly as a Smalltalk method, so it's natural for the semantics to implement it specially as well.

$$\textsc{New}\quad \frac{\begin{array}{c} fields(\sigma, x_2) = y_1, y_2, \cdots, y_n \\ O = x_2[y_1 := \texttt{nil}.y_2 := \texttt{nil}.\cdots y_n := \texttt{nil}] \\ \ell \text{ is a fresh label in } \Sigma \\ \sigma' = \sigma[x_1 \mapsto \ell] \qquad \Sigma' = \Sigma[\ell \mapsto O] \end{array}}{\langle \Sigma, \sigma, x_1 := x_2 \text{ new. } L\rangle \to \langle \Sigma', \sigma', L\rangle}$$

Whew! This might be the longest semantic rule we've written yet! But, it's fairly simple when we break it down into its component parts: The first premise says that the fields for x_1 are y_1 through y_n. The second premise defines an object with an appropriate shape for the class: it has all of the y_i fields, and each is given the value `nil`. `nil` is the default value of a field in Smalltalk. The third, fourth, and fifth premises add a link from the variable to a label and from that label to the object in the store and heap.

Next, we'll look at the `value` method of blocks. This is relatively straightforward:

$$\textsc{Block}\quad \frac{\sigma(y) = [L_1.z]}{\langle \Sigma, \sigma, x := y \text{ value. } L_2\rangle \to \langle \Sigma, \sigma, L_1.x := z.\ L_2\rangle}$$

Quite simply, to evaluate a block, we run its statements, and assign the value of its last statement—which, recall, we've restricted to being a single variable name—to the target of our assignment. Since a block contains a statement list, this operates over statement lists rather than statements.

Next, simple assignments of a variable to another variable:

$$\textsc{VarAssg}\ \frac{\sigma' = \sigma[x \mapsto \sigma(y)]}{\langle \Sigma, \sigma, x := y.\ L\rangle \to \langle \Sigma, \sigma', L\rangle}$$

That is, we can assign from a variable y to a variable x by looking up y's value in σ, and adding the same value mapped from x.

Next, fields:

$$\textsc{FieldRead}\ \frac{\sigma(x_2) = \ell \qquad \Sigma(\ell) = x_3[\cdots y := w \cdots] \qquad \sigma' = \sigma[x_1 \mapsto w]}{\langle \Sigma, \sigma, x_1 := x_2\texttt{->}y.\ L\rangle \to \langle \Sigma, \sigma', L\rangle}$$

$$\textsc{FieldWrite}\ \frac{\begin{array}{c}\sigma(x_1) = \ell \qquad \sigma(x_2) = w \\ \Sigma(\ell) = x_3[z_1 := v_1.\ z_2 := v_2. \cdots y := v_m \cdots z_n := v_n] \\ \Sigma' = [\ell \mapsto x_3[z_1 := v_1.\ z_2 := v_2. \cdots y := w \cdots z_n := v_n]]\end{array}}{\langle \Sigma, \sigma, x_1\texttt{->}y := x_2.\ L\rangle \to \langle \Sigma', \sigma, L\rangle}$$

To read a field, we look up the object by name in σ, look up that label in Σ, and find the field-value pair matching the name y. We then update σ to map x_1 to that value. To write a field, we similarly look up the object and value in σ, and look up the object's label in Σ, but this time, we update Σ to have an identical object, but with the field-value pair for y replaced with the new value. In practice, of course, we wouldn't replace O with an identical object changed slightly, but simply update the one field in place.

All that remains (really!) is method calls. Calling a method is similar to calling a procedure from Module 7, with one major difference: we need to *find* the method. We will thus break down calling a method into a "finding" step and a calling step. To find a method, we need to walk up the superclasses of the target class until we find the named method. We'll handle those steps first, by defining a function $method(\sigma, x, y)$, which finds a method of x with the name y:

$$\frac{\sigma(x) = z \text{ subclass: } x[\cdots y[Q] \cdots]}{method(\sigma, x, y) = y[Q]}$$

$$\frac{\sigma(x_1) = x_2 \text{ subclass: } x_1[\cdots z_1[Q_1]\ z_2[Q_2] \cdots z_n[Q_n]] \qquad y \notin \{z_1, z_2, \cdots, z_n\}}{method(\sigma, x_1, y) = method(\sigma, x_2, y)}$$

The first rule extracts a method named y from the class x if it's present in the class. The second rule says that if y is *not* found in the class (x_1)—that is, it's not among the methods z_1 to z_n—then the superclass (x_2) should be searched. If the method isn't defined at all, then *method* has no definition, so our semantics will get stuck.

With *method* defined, we can now call methods similarly to the Simple Imperative Language. Like in Module 7, we will do this by freshening its variables to avoid conflict, and assume that $freshen$ is already defined. Importantly, $freshen$ also needs to freshen the name "`self`". Our method call rule is as follows:

$$\textsc{Call}\ \frac{\begin{array}{c}\sigma(x_2) = \ell \qquad \Sigma(\ell) = x_3[\cdots] \\ M = method(\sigma, x_3, y_1 : y_2 : \cdots y_n :) \\ S = freshen(M) \qquad x_4 = \texttt{self}\ S \\ M\ S = y_1 : w_1\ y_2 : w_2 \cdots y_n : w_n[L_1.\hat{}w_r]\end{array}}{\begin{array}{c}\langle \Sigma, \sigma, x_1 := x_2\ y_1 : z_1\ y_2 : z_2 \cdots y_n : z_n.\ L_2\rangle \to \\ \langle \Sigma, \sigma, x_4 := \ell.\ w_1 := z_1.\ w_2 := z_2. \cdots w_n := z_n.\ L_1.\ x_1 := w_r.\ L_2.\rangle\end{array}}$$

Yes, it's another monster of a rule. But again, we can break it down into simpler parts: The first premise says that x_2 must refer to a label, and the second premise says that that label must refer in the heap to an object of some class x_3. The third premise uses that class name, x_3, to look up the method. Remember that methods in Smalltalk are named in several broken parts, in this case y_1 :, y_2 :, etc, so $y_1 : y_2 : \cdots y_n :$. The fourth premise generates a substitution to freshen the names in the method, and the fifth premise freshens the important name `self`. Finally, the sixth premise defines all the names in the freshened method, so that we can use them to take a step. In the conclusion, like in the Simple Imperative Language, we rewrite the method as writes to the freshened variable names corresponding to its arguments, then the method body. SIL didn't have returns, so we also include a write of the return value to the target variable.

The ubiquity of objects makes these the most complex rules we've seen yet, but if you've understood the behavior of Σ and σ, you should be able to disentangle them. Alternatively, if you've written in an object-oriented programming language, start from there. In particular, it's surprising that our semantics have no conditions or loops, but in fact, they're not needed: we can build them out of `True` and `False` classes exactly like Smalltalk does:

```
Object subclass: Boolean [
    " ... "
]

Boolean subclass: True [
    ifTrue: block [ | x | x := block value. ^x ]
    ifFalse: block [ ^nil ]
    ifTrue: tBlock ifFalse: fBlock [ | x | x := tBlock value. ^x ]
]

Boolean subclass: False [
    ifTrue: block [ ^nil ]
    ifFalse: block [ | x | x := block value. ^x ]
    ifTrue: tBlock ifFalse: fBlock [ | x | x := fBlock value. ^x ]
]
```

5 Inclusion Polymorphism

In our brief introduction to polymorphism, we mentioned *inclusion polymorphism*, but left it for later elaboration. In the context of object-oriented languages, the most common form of polymorphism is inclusion polymorphism. In this section, we discuss inclusion polymorphism and its relationship with inheritance.

Inclusion polymorphism is based on the arrangement of the types into a hierarchy of *subtypes*. A language may define subtypes however it wishes, but as a minimum restriction, a type `a` is a subtype of a type `b` if a value of type `a` can be used anywhere that a value of type `b` can be used. You may notice an analogy to inheritance here: if classes are our types, and this is the only restriction we place upon subtypes, then `Square` is a subtype of `Rectangle`.

We define subtyping formally with a a relation $<:$, which is reflexive and transitive, and is typically a partial order. That is, every type is a subtype of itself ($\forall \tau_1 . \tau_1 <: \tau_1$); if $\tau_1 <: \tau_2$ and $\tau_2 <: \tau_3$, then $\tau_1 <: \tau_3$; and there exist pairs of types τ_1 and τ_2 for which neither $\tau_1 <: \tau_2$ nor $\tau_2 <: \tau_1$. When types τ_1 and τ_2 are related by $\tau_1 <: \tau_2$, we say that τ_1 is a subtype of τ_2, and equivalently, that τ_2 is a supertype of τ_1. The goal of an inclusion-polymorphism-based type system is to assure that if $\tau_1 <: \tau_2$, then a value of type τ_1 can be used anywhere where a value of type τ_2 is expected, but the reverse does not hold.

The key type rule associated with inclusion polymorphism is known as *subsumption*, which can be written like so:

$$\text{T\_Subsumption} \quad \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

In words, an expression of type τ_1 may be given type τ_2 whenever τ_1 is a subtype of τ_2. Typically, T_Subsumption isn't written quite so generally, because this version is not syntax-directed: if we judge e to have type τ_1, we could then decide arbitrarily to give it *any* supertype of τ_1 in place of τ_1. Instead, subsumption is often written implicitly with respect to particular rules. For instance, consider a subsumptive version of a rule for adding integers:

$$\text{T\_AddSubs}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \text{int} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

Of course, a particular language might have a more precise rule that allows addition to yield a subtype of the integers, but this rule demonstrates that syntax-directed subsumption is possible. It remains to determine which types may be related by <:.

There are, broadly, two answers to this question: *structural subtyping* and *nominal subtyping*. We will focus on structural subtyping first.

5.1 Structural Subtyping

Consider these two Smalltalk classes, which we will define without any explicit superclass (which isn't actually valid in Smalltalk):

```
HelloEN [
    greeting ^<String> [ ... ]
]
HelloFR [
    valediction ^<String> [ ... ]
    greeting ^<String> [ ... ]
]
```

Both of these classes have no fields, and have a method named `greeting`. `HelloFR` additionally has a method `valediction`. The `greeting` methods have a lot in common, which we call their *signature*. The signature of a method is its name, number and type of arguments, and type of return. Given correct objects to call them on, two methods with the same signature are interchangeable; either will work in all the same situations. Since `greeting` was the only method of `HelloEN`, anywhere where a `HelloEN` can be used, a `HelloFR` can also be used. We haven't explicitly defined these classes as having any relationship to each other, but implicitly we can see that they meet our minimum bar for subtyping: that one can be used in place of the other.

> **Aside:** "Valediction" is to "greeting" as "goodbye" is to "hello": it's a word for words and phrases of parting.

A language in which that minimum bar is the *only* bar is said to employ *structural subtyping*. Assuming that the *signature* function in our formal model extracts the signature of a method, we can write a rule for subtyping. In the following rule, let the metavariables F and M range over field lists and methods, respectively, and assume that our class syntax has no explicit subtyping specified:

$$\text{T\_Structural}\ \frac{\begin{array}{c}\tau_1 = x[F_1 M_{1,1} M_{1,2} \cdots M_{1,n}] \qquad \tau_2 = y[F_2 M_{2,1} M_{2,2} \cdots M_{2,m}] \\ \forall v \in (1,n).\exists w \in (1,m).\mathit{signature}(M_{1,v}) = \mathit{signature}(M_{2,w})\end{array}}{\tau_2 <: \tau_1}$$

We can break down this rule into several parts. The first and second premise simply state that τ_1 and τ_2 are types referring to classes x and y, respectively. Class x contains methods $M_{1,1}$ to $M_{1,n}$, and class y contains methods $M_{2,1}$ to $M_{2,m}$. The third premise is the important one: if, for every method $M_{1,*}$, there exists a method $M_{2,*}$ with the same signature, then τ_2 is a subtype of τ_1. The particular quantifications are very important here. *Every* method in τ_1 must have a corresponding method in τ_2, but the reverse is not true. Thus, a τ_2 may be used in place of a τ_1, but a τ_1 cannot (necessarily) be used in place of a τ_2. Note that we haven't discussed fields; we'll get to why when we discuss encapsulation, in Section **??**.

Structural subtyping is uncommon in practice, for pragmatic reasons of implementation which we will address when we discuss nominal typing. Structural subtyping is sometimes also called *duck typing*—if it looks like a duck and quacks like a duck, it's a duck—but "duck typing" is more frequently used to describe an informal sense of types in a dynamically typed language than a static type system. A static type system can certainly be structural, but would usually not be called duck-typed.

Structural subtyping is distinct from parametric polymorphism by the presence of a hierarchy. In parametric polymorphism, a function with type parameter α must work for any substitution of α. Thus, it's more common

to use parameters in which α forms part of some larger type, such as $\alpha \to \alpha$, since α is essentially a black box, and $\alpha \to \alpha$ can at least do something. In a structurally subtyped inclusion-polymorphic language, if a method's parameter has type τ, it may get a value of a subtype of τ, but it can still count on having a definition for τ that holds.

The $<:$ relationship in structural subtyping is naturally reflexive and transitive. τ_1 will definitely have the same methods as τ_1 (reflexivity). If τ_3 has all the methods of τ_2, and τ_2 has all the methods of τ_1, then τ_3 clearly has all the methods of τ_1 (transitivity). However, we do not need explicit rules for reflexivity and transitivity, as these arise naturally from the T_Structural rule.

5.2 Nominal Subtyping

When we looked at structural typing, we left out any explicit specification of subclasses. In *nominal subtyping*, that explicit specification *is* the subtyping relationship. That is, $\tau_1 <: \tau_2$ if τ_1 was explicitly specified to be a subclass of τ_2. The only other type rules needed are reflexivity and transitivity, which must be explicitly specified in nominal type judgments.

For nominal typing to make any sense, we must have a way to explicitly declare subtype relationships, and as we already saw, Smalltalk has this: every class is declared as a *subclass* of another class, and if class a is a subclass of class b, then the type represented by a is a subtype of the type represented by b. Like with subtypes and supertypes, if a is a subclass of b, we say that b is a superclass of a. We don't define the superclass relationship as reflexive, however: a class is not a subclass of itself. For instance, we could declare several classes like so:

```
1 Object subclass: Greeter [
2     greeting ^<String> [ ... ]
3 ]
4
5 Greeter subclass: Parter [
6     valediction ^<String> [ ... ]
7 ]
```

We haven't explicitly given `Parter` the structure of `Greeter`, but it *implicitly* has the structure of `Greeter` because of inheritance, so the basic requirement of subtyping is satisfied: a `Parter` can take the place of a `Greeter`. Inheritance automatically gave us a sufficient relationship for subtyping, so nominal subtyping is simply using the inheritance relationship as the subtyping relationship.

Exercise 2. Write the rules for $<:$ in nominally typed Smalltalk. Remember, $<:$ must be transitive and reflexive!

We've said that classes are defined as subclasses of other classes, but this leaves a question: how do we define our first class? Object-oriented languages have two answers:

1. In many languages in which classes are "bolted on" to an existing imperative language, such as C++, classes may be defined without any superclass. These classes inherit nothing, and as a consequence, there are no methods shared by every class in the entire system.

2. In most languages which were designed to be object oriented from the beginning, such as Smalltalk and Java, there is a single class, typically called `Object`, which is defined a priori by the language implementation. There is no way for a programmer to define a class without a superclass, so the `Object` class is unique, and cannot actually be written in the language. This allows the language designer to ensure that some methods are available on every object, such as GNU Smalltalk's `display` method.

Nominal subtyping is usually justified by implementation concerns, rather than theory, as it's strictly more restrictive than structural subtyping. So, let's discuss those implementation concerns.

When we compile Smalltalk code, we have to decide how we're going to call methods on objects. Thus, we have to have some way, given an object, to find a given method for that object. One solution would be to remember every method's name, and use a hash map stored in the object to map every method name to its machine code. This is viable, but it can be too slow, or more importantly, its performance can be too unpredictable for most language implementations.

Instead, classes are compiled into *virtual tables* (also called vtables or vtbls), which are simply arrays of pointers to machine code. Every method gets an index in this array, typically simply in the order that they appear in the source code. For instance, the `Greeter` class above would have a virtual table with one element, which points to the machine code for `greeting`. If we know from our type judgment that a given value has `Greeter` as its type, then we know which element of its virtual table corresponds to `greeting`. Without subtyping, this is sufficient: remember, a type is just a set of values, so the `Greeter` type is all objects created as `Greeter`s, and all `Greeter`s will naturally be created with `Greeter`'s virtual table.

To support subclasses, all we need to do is make a subclass's virtual table compatible with its immediate superclass's virtual table. Since it inherited the methods anyway, that's simple: the virtual table for `Parter` has two elements, of which the first points to `greeting`, and the second points to `valediction`. Now, every `Parter` behaves like a `Greeter`, because the first element of its virtual table is a `greeting` method, and a `Parter` simply has a second method in an index that would never be used by a simple `Greeter`. If we had overridden `Parter`'s implementation of `greeting`, we would put that in the first slot of the virtual table, regardless of the order in which it was defined, to guarantee compatibility with `Greeter`.

This virtual table design works well, but makes structural subtyping essentially impossible. The compatibility between two virtual tables depends not just on the types of all the methods of their classes, but the order in which they happen to be declared, which is irrelevant for structural subtyping. For instance, `HelloEN` and `HelloFR` would have incompatible virtual tables. Thus, the desire for well-performing implementations spurred on a desire for nominal subtyping.

It is not impossible to implement similar optimizations in a structurally typed language—indeed, this is done for modern dynamic languages such as JavaScript, and for structural subsets of languages like Java—thanks to just-in-time compilation (JIT) and type profiling, but these are beyond the scope of this course.

> **Aside:** While beyond the scope of this course, JIT compilation is the instructor's area of expertise, so if you want to know more, he'd be happy to spend countless hours telling you more.

Although the most frequent argument for nominal typing is practical, there is also an argument about usability: structural subtyping can sometimes render accidental relationships between classes. In nominal typing, subtyping is never an accident, since it's always explicitly specified.

6 Smalltalk's Weird Methods

Smalltalk has a very unusual method syntax. However, it's superficial: there's nothing about its unusual method syntax that makes methods behave any differently in Smalltalk than they do in any other programming language.

Methods in Smalltalk take three forms: nullary methods (taking no arguments), operators, and n-ary methods.

Nullary methods are simply named by an identifier, such as `area`, and are called by placing the name next to the target object, such as `x area`.

Operators are named by symbols, but are otherwise just methods. In Smalltalk, `2 * 2` is a call to the `*` method on the `Number` object 2, with argument 2.

The signature of an n-array methods in Smalltalk is given with a sequence of pairs of a partial method name and a parameter name. For instance, the `setWidth: w setHeight: h` signature is for a method with name `setWidth:setHeight:`, and parameters `w` and `h`. In a language like Java, this method would probably be written `setWidthAndHeight(x, y)`, but the difference is superficial. When Smalltalk code is compiled, it shuffles these parts around to recognize the name `setWidth:setHeight:`, and that is (usually) used internally as the method's name.

As specified all the way back in Module 1, we can also write `Rectangle>>setWidth:setHeight:` to indicate specifically the `setWidth:setHeight:` method of `Rectangle`.

7 Subtyping and Methods

We haven't yet mentioned method types. At a basic level, method types are exactly the same as procedure types: a list of parameter types. Methods are procedures which return values (typically), so in addition, we need a return type. Thus, we can write a procedure type as $(\tau_1, \tau_2, \cdots, \tau_n) \to \tau_r$, where τ_1 through τ_n are the parameter types, and τ_r is its return type.

With non-object-oriented procedural languages, one could imagine a procedure being curried (and thus represented as $\tau_1 \to \tau_2 \to \cdots \to \tau_r$ instead), though very few, if any, languages like this actually exist[5]. With methods, the same does not apply, because *methods are not values*.

A method is called *on* an object. That object cannot then return a method, because that method would be naked; it would have no object to call it on. Many object-oriented languages support functions, so that those functions can be values when methods cannot; but, many don't.

The fact that methods are not values makes method types a bit of a dubious proposition. We said that types are sets of values, but now we've said that methods aren't values at all. In fact, method types aren't complete types: they only exist as part of object types. An object's type is defined by its class, which contains methods; no expression can have a method type, but method types nonetheless exist, bound up within object types.

Method types themselves aren't complicated, and are usually written explicitly. For instance, the `Rectangle>>setWidth:` method would be written in Strongtalk as

```
setWidth: v <Number> ^<Rectangle> [ ... ]
```

Smalltalk has no unit type, so the typical return type from a method that doesn't need to return anything is the type of the surrounding class, and the default return value is `self`. This method's type is `(Number)` → `Rectangle`.

Note that the hidden parameter—`self`—does not represent any part of this type! That's because it doesn't need to: remember that method types are part of their surrounding object types, so the type of the hidden parameter is implied. There are no naked methods.

Now, let's look at `Square>>setWidth:`. It would be written in Strongtalk as

```
setWidth: v <Number> ^<Square> [ ... ]
```

This method's type is `(Number)` → `Square`. But wait, `Rectangle>>setWidth:`'s type was `(Number)` → `Rectangle`, and that's not quite the same. We said that subtypes would depend on the signatures of the methods being the same, but this signature is not the same! We need a concept of method compatibility, and with it, method subtyping.

The goal of subtyping is that if $\tau_1 <: \tau_2$, then a value of type τ_1 can be used in the place of a value of type τ_2. The same is true of method subtyping: we want a method type τ_1 to be a subtype of τ_2 if τ_1 can be used in place of τ_2. But, when is this true?

First, let's consider the return. `Square>>setWidth:` returns a `Square`, but `Rectangle>>setWidth:` was expected to return a `Rectangle`. This isn't actually a problem, though: `Square`s are `Rectangle`s! The return type is compatible so long as it's a subtype, so we can fill in part of our subtyping relationship:

$$\frac{??? \qquad \tau_{1,r} <: \tau_{2,r}}{(\tau_{1,1}, \tau_{1,2}, \cdots, \tau_{1,n}) \to \tau_{1,r} <: (\tau_{2,1}, \tau_{2,2}, \cdots, \tau_{2,n}) \to \tau_{2,r}}$$

For a method type to be a subtype, its return type must be a subtype.

Now, let's consider the parameters. At a bare minimum, we can insist that the *number* of parameters must be the same. In fact, in Smalltalk, it's not even possible for the *name* to be the same if the number of parameters isn't the same. Thus, we need to consider how we're allowed to override a given parameter type.

Consider overriding `Square>>setWidth:` to take an `Object` as its argument instead of a `Number`. `Object` is the top of our type hierarchy, so `Number <: Object`. As a consequence, this wouldn't break anything: we can still

[5] I don't know *why* few languages like this exist, but to proffer a guess, it's probably because currying makes it unclear when a function is actually run, but that doesn't matter in a referentially transparent language. Procedural languages are not generally referentially transparent, so it's very important to be abundantly clear when a procedure is actually executed.

take all the argument values that `Rectangle>>setWidth:` can take, we can just accept more. The reverse, however, isn't true: if `Square>>setWidth:` took `Float` as its argument (`Float <: Number`), then we wouldn't be able to use `Square>>setWidth:` in all the ways we could use `Rectangle:setWidth:` (i.e., we couldn't pass it any other number type), so we can't allow a subtype. What's surprising about this, however, it that it's in reverse. It was safe for `Square>>setWidth:`, in a subtype of `Rectangle>>setWidth:`, to take a *supertype* of `Number` as its argument!

With this surprise, we may now fill in the rest of our subtyping relationship:

$$\frac{\forall v \in (1, N). \tau_{2,v} <: \tau_{1,v} \qquad \tau_{1,r} <: \tau_{2,r}}{(\tau_{1,1}, \tau_{1,2}, \cdots, \tau_{1,n}) \to \tau_{1,r} <: (\tau_{2,1}, \tau_{2,2}, \cdots, \tau_{2,n}) \to \tau_{2,r}}$$

Note how the type requirement for the parameters is the inverse of the requirement for the return: the parameter types on the right must be subtypes of the parameter types on the left, while the return type on the left must be a subtype of the return type on the right. This reversal of the subtyping relationship of embedded types within a constructed type is called *contravariance*, and when the relationship is not reversed, we call it *covariance*. An additional possibility is *invariance*, which we will discuss in Section **??**.

There's one final corner to discuss: What about that hidden parameter, `self`? The type of `self` in `Square>>setWidth:` is `Square`, since `Square>>setWidth:` will always be called on `Square`s. Similarly, the type of `self` in `Rectangle>>setWidth:` is `Rectangle`. So, the hidden parameter type is covariant, even though the other parameter types are contravariant! In fact, this corner is why it's often confusing to think of the target as a hidden parameter. The reason we can confidently define `self`'s type as `Square` within `Square>>setWidth:` is precisely because it's not actually an argument, but the target: the very fact that we found this method means `self` must have been a `Square`. This is also why it's necessary that methods cannot be values: if you were to strip a method from its object and call it on some other object, you would need to know, for instance, that you'd stripped it from a `Square`, even if the type system said it was a `Rectangle`. This odd typing anomaly of `self` itself is part of why the target is usually not included in a method's signature.

8 Fields and Encapsulation

One of the principles of object-oriented programming is that the details of how a particular object works should be hidden, so that those details can be changed without needing to change the *interface*. The interface is whatever part of an object is accessible from outside that object; but, what should it be? Generally speaking, an object's methods are accessible from outside the object, as otherwise they'd be rather useless. Many languages allow specific methods to be marked `private`, which makes them only accessible from other methods within the object.

But, what about fields?

First, let's consider the most open option, by making all fields accessible from outside the object. What this means is that objects truly behave like records, but with methods. This completely breaks our principle of hiding the implementation, since every aspect if it is bared to all users, but that's just a principle, not a rule.

If we do allow all fields to be fully accessible, what does this mean for typing objects? Let's expand our `Rectangle` type from above to be explicit about the `width` and `height` fields:

```
Object subclass: Rectangle [
    | width <Number> height <Number> |
    ...
]
```

We don't need to add any fields to `Square`, because it inherits these fields from `Rectangle`. And, let's imagine an expanded Strongtalk syntax that allows you to access fields of objects other than `self` using an ASCII arrow, e.g., `r->width`. So, we could forcibly set the width of a rectangle (or square!) to 10 with `r->width := 10`, and get the width directly.

Now, let's consider a new class, `IntSquare`. In Strongtalk (and Smalltalk), `Integer` is a subtype of `Number`, so we might quite reasonably want a square with integer width and height, an `IntSquare`, to be a subclass of `Square`. We define it as follows, assuming that in this unlikely version of Strongtalk, we can override fields in the same way that we override methods:

```
Square subclass: IntSquare [
    | width <Integer> height <Integer> |
    ...
]
```

And now, let's look at an unrelated method that expands a rectangle by a factor of 1.5:

```
expandRectangle: r <Rectangle> ^<Rectangle> [
    r->width := r->width * 1.5.
    r->height := r->height * 1.5.
    ^r
]
```

But now, we have a problem. What happens if we call the `expandRectangle:` method with an `IntSquare` as its argument? We'll attempt to set `width` to a floating-point number (probably), but this doesn't work; an `IntSquare`'s `width` field is of type `Integer`! So, we can't possibly let field overrides be covariant.

Well, if field overrides can't be covariant, how about contravariant? Let's imagine a new kind of square, that's not as restrictive about the sides being defined by boring numbers. `AbstractSquare` will allow the sides to be of any object type (but don't ask what this actually *means*):

```
Square subclass: AbstractSquare [
    | width <Object> height <Object> |
]
```

Unfortunately, this still doesn't work. If we pass an `AbstractSquare` to `expandRectangle:`, it attempts to call the `*` method on an `Object`, and so fails.

In fact, the only way we can define a `width` field in the subclass is with exactly the same type as in the superclass. We call this requirement type *invariance*, and because of it, there's no reason to allow field overrides at all: if you can only "override" the field by declaring an identical field, then there's no real overriding being done.

Invariance also comes up if we have OCaml-style references in a language with method (or function) subtyping: `ref` τ_1 is only a subtype of `ref` τ_2 if $\tau_1 = \tau_2$.

Exercise 3. Work out why OCaml-style references require invariant typing. You may want to imagine a reference class with methods `put` and `get`.

Recall that this entire discussion was in the context that we make fields completely open. Smalltalk takes the most opposite approach: fields are completely private; you can't even access the fields of a superclass in a subclass! This fits the design goal of hiding implementation details well, since even if you inherit from a class, you can't touch its fields, but it's also fairly inflexible. Because fields are private to particular classes, you can even define fields with the same *name* in a subclass. The field doesn't override, it's just a totally distinct field: `Rectangle>>width` is a different field from `IntSquare>>width`, and when a method in `Rectangle` reads `width`, it will only find its own.

Weirdly, although Smalltalk fields are as private as possible, Smalltalk doesn't even support private methods. All methods in Smalltalk are usable by anyone with a reference to the object. Software which intends to be more controlled works around this by defining an interface class (sometimes called a proxy class) and an implementation class for all public types, where the interface class has a field reference to the implementation class, and exposes only the methods intended to be public.

Most programming languages choose a compromise somewhere between Smalltalk and our hypothetical all-open language, allowing individual fields and methods to be declared private or public.

Because, as we've now said many times, the purpose of the subtyping relationship $\tau_1 <: \tau_2$ is to ensure that a τ_1 can be used anywhere where a τ_2 is expected, field and method privacy also affects subtyping. The effect is quite simple, though: we can simply disregard all private fields and methods, since they don't affect the public interface of objects. This is also why we disregarded fields entirely while describing `<:`: fields are completely private in Smalltalk.

9 Overloading

In Smalltalk, an object may have two fields with the same name. This is possible if the fields are defined on different classes, and works because it's completely unambiguous which field is meant in any context, since methods of a class may only access the class's own fields, and not any superclass's fields. This is one kind of *overloading*. Most statically typed object-oriented languages also allow method overloading. Strongtalk does not, but we will use its syntax to describe the idea.

Broadly, overloading is a facility by which different entities in the same context can share the same name. Overloading is one (very limited) form of polymorphism, so an object-oriented language with overloading is mainly inclusion-polymorphic, and secondarily overloading-polymorphic. References to the shared name are disambiguated by information from the context of the reference. In the case of overloaded methods, the compiler examines the number and type of the arguments passed to the method, and in some languages, also the return type of the method. It then selects the instance of the shared name that best fits. If there is no match, or if there are multiple matches, the compiler signals an error. The process of disambiguating references to overloaded names is known as *overload resolution*.

In Smalltalk, the number of arguments to a method is part of the name of the method, so it's meaningless for two methods with the same name to have a different number of arguments. However, we could imagine defining methods with the same name but different argument types.

Consider a Strongtalk class for handling money in dollars and cents, with a method for adding two dollar amounts together:

```
Object subclass: Money [
    | dollars <Integer> cents <Integer> |

    Money class >> withDollars: d cents: c [
        " ... constructor ... "
    ]

    dollars ^<Integer> [ ^dollars ]
    cents ^<Integer> [ ^cents ]

    + second <Money> ^<Money> [
        | d c |
        d := dollars + second dollars.
        c := cents + second cents.
        c > 100 ifTrue: [
            d := d + 1.
            c := c - 100.
        ].
        ^Money withDollars: d cents: c
    ]
]
```

For pragmatic reasons, it may also be useful to add an amount of money specified as a number. With support for method overloading, we could simply define a second `+` method, this time taking a `Number` as an argument:

```
...
    + second <Number> ^<Money> [
        " ... round second's cents, perform the addition, etc... "
    ]
...
```

Now, consider the expression `m + 42`. This is a call to the `+` method on `m` (which we will assume is a `Money`), with `42` as an argument. Assuming the compiler is doing its job, it should know the types of `m` and `42`: `Money` and `Integer`, respectively. Since `m`'s type is an object type—indeed, all values are objects in Smalltalk—its type is defined by a class. Looking into this class, the compiler finds two `+` methods: one expecting a `Money` as an argument, and one expecting a `Number`. `42` is neither a `Money` nor a `Number`, which on the surface is a problem. An ideal method would be of some type `(Integer) → Object` (`Object` because in this context, we don't care what the return type is), but our options are `(Money) → Money` and `(Number) → Money`. Subtyping can rescue us again: we need a method that can take an `Integer` as an argument, so we need a method with an argument type which is a supertype of `Integer`. Indeed, we need a method which is itself a subtype of our ideal method type! `(Number) → Money <: (Integer) → Object`, so we select that version of the method.

Overload resolution can get vastly more complicated than this. For instance, if we had a version of `+` for `Number`, but a third version of `+` for `Integer` specifically, then both of those methods fit a call with an `Integer` argument. In some languages, this ambiguity is simply disallowed, or the call is not allowed if it has multiple matches. In other languages, such as C++, there is an algorithm by which a "closest match" is chosen, but that algorithm is not always intuitive or obvious.

Method overloading introduces several complications which may not be immediately obvious. Almost without realizing it, we've lost a useful feature: erasability! With System F, for example, we could erase all of our types, and our semantics would behave exactly the same. Until method overloading, the same was true of procedural and object-oriented types. This is so because while classes *define* types, classes are *not* types in and of themselves; all type declarations can still be erased. Indeed, erasing all type declarations from Strongtalk yields Smalltalk, and that fact is part of why Strongtalk doesn't allow method overloading.

With method overloading, types actually affect compiled code: we need to know which overloaded method to call, and the only way to decide is with the types. This causes difficulty both for our formal semantics, which don't usually mix types and semantics, and for a language implementation, which often needs to have a unique name for a method simply to generate code. We typically resolve this in our formal semantics by ignoring it: for all its usability benefits, method overloading isn't semantically very interesting, so is typically left out of semantics for object-oriented languages. In implementations, this is resolved by having a type-directed compilation step called *name mangling*.

Name mangling is simply a renaming stage in which methods (or anything else with overriding allowing conflicts) are renamed to system-generated names containing their surrounding class's name, the types of all of the arguments, and, if it's used in overloading, the return type. In fact, every time we've specified a field or method with `>>`, we've been performing a light form of name mangling. A Strongtalk-with-overloading compiler could rename `+` on numbers as, for instance, `Money>>+<Number>`. Name mangling may also involve reducing the character set of mangled names, usually for platform-specific reasons such as label names in assembly code, in which case `+`'s name could instead become, say, `_ZN5MoneyplE6Number`. It's not worth trying to disentangle this particular name (it was generated by GNU C++); it's just an example of character-set-reduced name mangling.

9.1 Parametric and Overloading Polymorphism

We've only mentioned overloading polymorphism now, in the object-orientation module. Overloading is perfectly compatible with imperative languages, as well, and many imperative languages allow procedure overloading.

Parametric polymorphism and overloading polymorphism aren't inherently incompatible, but they don't work well together. For instance, in Java, a generic class cannot overload `Object` and the type parameter in the same method, because the resulting rules on choosing and overload would be ambiguous.

However, there is a more pressing reason why most parametric polymorphic languages don't have overloading: most parametric polymorphic languages are curried, and currying is difficult or impossible to reconcile with overloading. If a function is overloaded on its second parameter, but the language is curried, then there must be some way to pass in the first argument and get a function. That function would have an unresolved overload, since the second, type-overloaded argument hasn't been given yet. An unresolved overload is a set of functions, not a function, so it's not a value. The same problem arose with type inference: an unresolved polymorphic type cannot be a value.

10 Completing The Type Hierarchy

Now that our types form subtypes in a hierarchy, we can discuss the extreme points of that hierarchy.

We've already talked about `Object`, the root of the hierarchy. In Smalltalk, because every class must be declared as a subclass of some other class except for `Object`, every class is, unavoidably, a subclass (directly or indirectly) of `Object`, and thus every value is a subtype of `Object`. `Object` forms the top of our type hierarchy, as $\forall C.C <:$ `Object`. The top of a type hierarchy is written as $\top$, which is not a capital T, because mathematicians love creating as many nearly-indistinguishable symbols as possible.

Many object-oriented languages do not have `Object` as the root of their hierarchy, and many mix objects with other kinds of data. For instance, in C++, there is no root of the class hierarchy, and in both C++ and Java, simple primitive values such as `int`s aren't objects at all. Even in a language like Smalltalk, where numbers *are* objects, to define the semantics of numbers, we need to eventually describe them in terms of actual, mathematical numbers, which are not objects. In these cases, is there a top of the hierarchy? Should there be?

> **Aside:** As an implementation detail, no Smalltalk implementation actually allocates objects for most numbers used in Smalltalk, and so the actual value at run-time is not a reference to any object. Yet, the language still acts as if they're references. Since object-oriented languages such as Smalltalk hide the implementation details of objects regardless, it's possible to have the user language remain purely OO but implement faster numbers as a transparent optimization.

C++'s answer is no. As a consequence, on the surface, it may seem impossible to write a C++ procedure which accepts any value as an argument, but in fact, C++ implements a form of parametric polymorphism called *templates* to get around this problem. We'll talk a bit about templates in Section **??**.

Java uses another form of polymorphism to paper over the problem of $\top$: *coercion*. Coercion is an ad hoc form of polymorphism in which values of some types are allowed to take the place of values of another type by converting them at run-time. For instance, in Smalltalk, if you add `3 + 0.14`, because the underlying machine can only add together two integers or two floating-point numbers, the integer `3` is first converted into a floating-point value. In Java, as well as numeric type coercions like that, all non-object types are coercible into object types. For instance, if you write `Object o = 42;` in Java, then it will insert code to create an instance of the `Integer` class, which is a subclass of `Object`. In this way, even though `int` is not explicitly a subclass of `Object`, an `int` can be used anywhere an `Object` is expected, by coercion. By defining $<:$ with both nominal types and coercion in mind, we restore `Object` as the root of the type hierarchy. Note that coercion is not erasable.

> **Aside:** Java also implements overloading, so Java has inclusion polymorphism, overloading polymorphism, *and* coercion polymorphism! If you're keeping score, that's three of the four forms of polymorphism from the Cardelli-Wegner polymorphism hierarchy discussed in Module 4. Now, if only we can get parametric polymorphism too...

We've found the top of the type hierarchy. Is there a bottom? That is, is there some type τ which is the subtype of every possible type? Again, different languages have different answers; in fact, it's even more divisive than the top type. In any case, the bottom type is written $\bot$.

One answer is that bottom is the type of errors, exceptions, failures, and infinite loops. That is, it's the type we give to an expression that will not produce a value, or will not produce a "normal" value. This is Haskell's answer, and fits typing well, because it allows that any expression could have such an error.

A second answer is simply that there needn't be a bottom type. With inclusion polymorphism, the bottom type is odd anyway: it would need to be able to take the place of any type in the entire system, and thus would presumably have to have every defined method, and more generally, every conceivable method. Given that a value cannot have every conceivable method, we can simply reject the idea of a bottom type. The only problem with this answer is the initialization problem from Module 7: if an object of a class has all the fields defined in that class, what are their values before you've first assigned them? With object orientation, this problem only gets more difficult to solve: how can we set up a complicated object hierarchy with interactive, even cyclic, object references, if we're also not allowed to set an object's fields directly? There is no satisfying answer to this question, and it remains the subject of active research.

The third answer is both the most controversial and the most popular: `null`.

> "I call it my billion-dollar mistake. It was the invention of the null reference in 1965."
>
> — Tony Hoare

The null reference is written in different ways in different languages: `nullptr`, `null`, `nil`, etc. We'll use `nil`, because that's Smalltalk's name for it. `nil` is both a type and a value, and there is only one value of type `nil`: `nil`.

`nil` is an object with no methods and no fields[6]. And $\forall\tau$. `nil` $<: \tau$. The fact that `nil` is a subtype of every other type doesn't arise from the typing rules; indeed, it makes no sense! How can a type with no interface be a subtype of every other type? The answer is, because it's an axiom:

$$\text{T\_Nil}\ \frac{}{\texttt{nil} <: \tau}$$

We simply allow `nil` to be taken as any type, by fiat.

This, of course, breaks the entire concept of typing. The goal of type checking is to prevent the semantics from getting stuck (and thus, presumably, prevent the implementation from crashing, or needing to consider badly-typed cases, etc), but how can the semantics possibly *not* get stuck if we try to call a method on a value that has no methods? The answer is, it gets stuck. And, the implementation usually crashes or throws an exception.

So, why add this problematic $\bot$ type, `nil`, if it breaks our type safety and causes crashes at run-time?

> "... At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."
>
> — Tony Hoare

An answer to this question that's less self-flagellating than Tony Hoare's is the initialization problem. `nil` gives us a powerful aid in initialization: a *default value*. Since `nil` is a subtype of every type, we can set every field to `nil` to start with, and leave the problem of correct initialization to the programmer!

An amusing second consequence of `nil` is that every proof of type safety of an object-oriented language with `nil` has an extra caveat: an expression will yield a value of the judged type, reduce forever, *or attempt to dereference null*. Oh well; except-for-null type safety is *almost* type safety I suppose.

11 Casting and Recovering Types

Notice that in our semantics, every object in the heap carries with it the name of its class. This is a form of *run-time type information*, and it allows us to recover type information. It is, of course, another way in which types are usually *not* erased in object-oriented languages.

For instance, consider the following snippet of Strongtalk code:

```
1 Object subclass: IntHider [
2     | value <Integer> |
3
4     add: x <Number> [
5         value := value + x.
6     ]
7 ]
```

This may seem fine, but actually contains a type error! We're adding an `Integer` to a `Number`, so the result is a `Number`, not an `Integer`! In fact, because of Strongtalk's lack of overloading, this would occur even if `x` were defined as an `Integer`. Most object-oriented languages provide some kind of *run-time type checking* and *casting*, by which we can check if a value has a particular type, and ascribe it that type, respectively. Strongtalk doesn't, so we'll have to imagine syntax for it:

[6] Actually, in Smalltalk, `nil` is of the `Undefined` class, because Smalltalk wouldn't dare make anything *not* be a class, but this bit of knowledge is in no way useful to understanding `nil`.

```
Object subclass: IntHider [
    | value <Integer> |

    add: x <Number> [
        | v <Number> |
        v := value + x.
        v is<Integer> ifTrue: [
            value := v <Integer>.
        ] ifFalse: [
            self halt. " unrecoverable, crash "
        ].
    ]
]
```

`v` is a `Number`, so the assignment to `v` is allowed. The `is<Integer>` is an imagined syntax for checking if `v` is of the class `Integer`, and `v <Integer>` is a similarly imagined syntax for casting.

Type checking involves looking in the run-time type information to validate the type. Because of inheritance, this means looking through an entire chain of classes. This also means that some information needs to be stored about classes in the run-time system. In most object-oriented languages, `Class` is a class, and classes are elements of that class, so that the run-time system can look information like this up. Thus, the `Class` class contains a method something like this:

```
isSubclassOf: query <Class> [
    self = query ifTrue: [ ^true ].
    superclass isNil ifTrue: [ ^false ]. " we've reached the root of the object hierarchy "
    ^superclass isSubclassOf: query
]
```

Casting, which we've written as `v <Integer>`, gives the expression `v <Integer>` the type `Integer`, but at run-time, performs a check, so that the type is guaranteed to be correct. If the check succeeds, then this expression is the same as `v`; the cast doesn't do anything. If the check fails, though, the behavior depends on the language. Some languages evaluate to `nil`, since `nil` is of every type. Others raise an error, which yields yet another exception to type safety in object-oriented languages: an expression will yield a value of the judged type, reduce forever, attempt to dereference null, *or perform an incorrect cast.*

> Me: Can we get some type safety?
> Mom: We have type safety at home.
>
> Type safety at home:
>
> > "We prove a type soundness result for FJ: if a well-typed expression `e` reduces to a normal form [...] then the normal form is either a well-typed value [...] whose type is a subtype of the type of `e`, *or stuck at a failing typecast.*" — Featherweight Java**?**, emphasis added

Yet another solution is *flow typing*, which allows us to eschew casting entirely. If we examine the code with knowledge of how `is<Integer>` works, then it's clear that in the `ifTrue:` block, `v` *must* be an integer. A flow-typing-based type checker would give `v` the type `Integer` in that block, and the type `Number` everywhere else. Flow typing requires considering many more and more complex cases (consider loops, for instance), and is beyond the scope of this course.

12 Generics

Consider a class defining a linear linked list:

```
Object subclass: List [
    | el next |
    " constructor... "

    setEl: to [
        el := to.
    ]

    el [ ^el ]

    setNext: to [
        next := to.
    ]

    next [ ^next ]
]
```

If we want to give this class types, that's relatively easy for `next`—it's a `List`—but what about `el`? We could make `el` of type `Object`, and we would then be able to store anything in our list, but presumably anyone using the `List` type knew specifically what they were storing. Setting `el` to something so generic would force every user of the class to cast every time they retrieve an element from the list, which is unfortunate. If we make `el` a more specific type, then our list would only be useful for that type.

We've actually already seen the solution: parametric polymorphism. Recall that in System F, we can define a function over types, a Λ (capital lambda, second-order lambda) function. A Λ function isn't directly usable as a λ-calculus function, because a type must be supplied first. In object orientation, the same concept exists over classes: *generics*.

A generic is a family of classes, defined by a function over types. To get an actual class, you need to call that function with a type as its arguments. Like in System F, the syntax for these type functions is distinct from the syntax for normal functions: in Strongtalk, we declare and call a type function with square brackets. Let's rewrite `List` to use generics:

```
Object subclass: List[A] [
    | el <A> next <List[A]> |
    " constructor... "

    setEl: to <A> [
        el := to.
    ]

    el ^<A> [ ^el ]

    setNext: to <List[A]> [
        next := to.
    ]

    next ^<List[A]> [ ^next ]
]
```

`A` here is not a specific class, but *any* class; it's a type variable. `List` is not a class, but a family of classes. To get a specific class, we instantiate `List` with a specific type argument, such as `List[Integer]`. Like Λ-abstractions in System F, this is simply done by substitution, in this case substituting `A` for `Integer`, like so:

```
Object subclass: List[Integer] [
    | el <Integer> next <List[Integer]> |
    ...
]
```

Of course, one never explicitly writes the above code; it's merely the result of using `List[Integer]`. As a result, we only need to write the structure of a list once, and it is then instantiated into as many specific list types as the program needs. Note that since `List` is not itself a class, it isn't a type either. `List[Integer]` is a type.

Generics raise some type-theoretic issues. First, there's the question of when to type-check a generic class: we

can type check the class with no specific type for the parameter type, but then the parameter type is completely opaque. Or, we can type check it only when it is used with a specific type as argument, which allows it to be defined only for argument classes with the right interface, but means we can't type check it unless or until it's used.

The bigger question is, is `List[Integer]` <: `List[Number]`? It can't be, because that would allow the `setEl:` method to be used wrongly. Then, is `List[Number]` <: `List[Integer]`? Again, it can't be; in this case, `el` would return the wrong type. In most languages, generic types are invariant over their parameter types for this reason. Some languages instead allow parameter types to be declared as explicitly covariant or contravariant. An explicitly covariant parameter type cannot be used in a method argument, because its actual type may be a supertype of its specified type. An explicitly contravariant parameter type cannot be used in a method's return. Neither may be used as a field.

Generics are erasable, in that a generic family of classes can be replaced with a single class if all types are removed. However, types aren't generally erasable in object-oriented languages, so erasing generic types can cause problems. For reasons of backwards compatibility, Java's generic types are erased, but as a consequence, casts between different, e.g., `List` types are allowed, causing errors when they're used.

> **Aside:** If generics are a form of parametric polymorphism, and Java has generics... yes, that's right, Java has inclusion polymorphism, overloading polymorphism, coercion polymorphism *and* parametric polymorphism! That's all four forms in the Cardelli-Wegner polymorphism hierarchy, bingo! Presumably, someone at Sun asked "what sort of polymorphism should Java support?", and the response was "yes".

Generics are defined over a specific class, e.g., `List[A]`. If we extend it to any number of classes, or procedures, etc, we get *templates*, which are C++'s equivalent. Although this makes templates very powerful, they're actually exactly what we already had with Λ-abstractions; it's generics that restricted this. Thus, templates aren't distinct from our perspective, so we won't discuss them further. We can simply consider "template" as another name for "generic".

13 Multiple Inheritance

Some object-oriented languages permit *multiple inheritance*, by which a class may inherit code from more than one other class. For instance, we could imagine writing

```
(Money, Rectangle) subclass: Wallet [ ... ]
```

to create a `Wallet` class which is both a `Money` and a `Rectangle`.

Multiple inheritance poses semantic problems when a class inherits from two or more classes that define methods with the same name. Consider, for instance, adding `+` to `Rectangle`, so that we can add two rectangles together. Given `v1` of type `Wallet`, what is the meaning of `v1 + v2`? It could mean either. Overloading could provide a solution—after all, the two versions of `+` don't have the same argument type—but even that falls flat: what if `v2` is also a wallet? There is no general answer to this conundrum. Most languages ban multiple inheritance entirely because of it. Some make the conflicting method (in this case, `+`) unusable on the problematic class, requiring an explicit cast to differentiate which method is meant. Some have explicit ways of stating preferences. And some, including Strongtalk, have a specific ordering to multiple inheritance, with later superclasses having priority over earlier ones[7].

Another consequence of multiple inheritance is a phenomenon known as *repeated inheritance*. Suppose class `A` inherits from classes `B` and `C`, each of which also inherits from a class `D`, which has a method `g`. Then `A` inherits `g` twice: once through `B` and one through `C`. If `g` is not overridden in either `B` or `C`, then there is no problem; the two `g`s can be merged into one, as they're the same. On the other hand, if one of the classes, say `C`, overrides `g`, then which should we prefer? Worse yet, what if both override `g`?

As bad as multiple inheritance is on our semantics, it's much worse for implementation. Consider our virtual tables, from Section **??**. Virtual tables worked because subclasses could look like their parent classes in the first

[7] Actually, Strongtalk implements *mixins*, which aren't quite the same as multiply-inherited classes, but are close enough for our purposes.

elements of their virtual table, and then add their own methods afterwards. But, the virtual table for `Money` will want `dollars` to be in the first element, and the virtual table for `Rectangle` will want `setWidth:` to be in the first element, so how can we define a virtual table for `Wallet`? There are many solutions to this conundrum, and none are very good.

A simple solution is to disallow multiple inheritance.

Another solution is multiple virtual tables. Java classes have one virtual table for their class, and an extra virtual table for each *interface* they implement. In Java, an interface is a list of method signatures, with no bodies; formally, then a Java class doesn't *inherit* from an interface at all, since the interface doesn't define any behavior, which solves the name clash issue as well. The difficulty with multiple virtual tables is just looking them up. Classes in Java contain a hash-map mapping interfaces to their virtual tables, and several layers of optimization to make this a bit less terrible than it sounds.

Another solution is *sparse* virtual tables, used by many C++ compilers. If the compiler can see all of the classes it needs to compile, then it can anticipate the multiple-inheritance problem, and intentionally define the virtual tables for `Money` and `Rectangle` such that they don't conflict, by giving `Rectangle` enough unused elements at the beginning to fit a `Money` virtual table in as well, or vice-versa. Even in the best-case implementation, this will leave gaps in the virtual table of one or the other, which is inefficient. But, there are very few virtual tables for very many objects, so that inefficiency isn't very important.

14 Blocks

In order for a language to be Turing-complete, it must have a way to represent decisions, and a way to repeat. In structured imperative languages, this is the `if` statement. Smalltalk, in an effort to be more-object-oriented-than-thou, instead opted for *blocks*.

You should be familiar with the syntax of blocks, and from our semantics above, their semantics as well. Their semantics of blocks are unusual, and blocks always need to be specially handled, because they contain statements, but those statements are in the context of the surrounding method. Thus, a block is not like a method, because it's nested within another method.

In Smalltalk, all blocks are objects of the `Block` class, which has a field containing the internal representation of the block's code. Thanks to encapsulation, the details of how the statements are actually stored do not need to be exposed to the end user, similarly to the IO monad in Haskell.

Blocks may also define their own local variables, which raises the question, what happens if you evaluate a block within itself? Most modern Smalltalk implementations treat blocks like procedures, creating multiple instances of the variables the block contains (technically, multiple stack frames). But, this makes blocks slightly slower, since they need to allocate and free space; classic Smalltalk implementations have only one instance of a block's variable per call to the surrounding *method*. Thus, this method will have different effects depending on the version of Smalltalk:

```
| block |
block := [:then :x |
    then value: [] value: 5.
    x
].
(block value: block value: 42) displayNl.
```

If only one `x` is allocated, this method will print `5`, because the recursive call to itself has replaced the value of `x`. If an `x` is allocated for each call, this method will print `42`.

An even more unusual behavior of blocks is return statements. A block may contain return statements, and if one is encountered, it returns from the *surrounding method*. This is odd for two reasons. To understand the first, consider this example, in which we've explicitly defined `True` for context:

```
1  Boolean subtype: True [
2      ifTrue: block [
3          ^ block value
4      ]
5  ]
6
7  Object subclass: Foo [
8      foo [
9          true ifTrue: [
10             ^ 42
11         ]
12     ]
13 ]
```

When we call `Foo>>foo`, it calls `True>>ifTrue:` with the block on line 9 as an argument. `True>>ifTrue:`, in turn, calls `value`, evaluating the block. The stack now looks something like this:

```
Foo>>foo
True>>ifTrue:
block from Foo>>foo
```

When that block evaluates its return statement, `Foo>>foo` returns, bypassing `True>>ifTrue:` entirely! So, Smalltalk implementations need to be able to break out of multiple layers of the stack.

> **Aside:** For those curious about implementation, this is done by having block objects contain a stack location, and then (fairly brutishly) force the stack pointer to that location when they return. This technique wouldn't work in all languages; the alternative is to explicitly read and "unwind" the stack, which is what's done by exceptions in most languages that implement them.

Weirder still, since blocks are values, we can actually strip a block from its surrounding method, even if it contains a return statement:

```
Object subclass: Bar [
    bar [
        ^[ ^42 ]
    ]
]

Object subclass: Baf [
    baf [
        | x |
        x := Bar new.
        x := x bar.
        x value. " Where does this block return from??? "
    ]
]
```

There is no good answer to the question of where a block removed from its surrounding method returns. Most just raise an error. For instance, GNU Smalltalk will report:

```
Object: 42 error: return from a dead method context
```

15 Object-Based and Prototype-Based Languages

We've focused our attention on class-based languages, but a language does not need to be class-based to be object oriented. We now turn our attention from class-based languages to *object-based languages*. Object-based languages differ from class-based languages in that they lack an explicit "class" construction for defining groups of similar objects. Many of the characteristics typical of object-based languages are outlined in a document known as *The Treaty of Orlando* **?**.

Proponents of object-based languages argue that a class is a rigid structure that establishes a template for all future code reuse, thereby constraining the ways in which software may evolve. Thus, proper use of classes requires some degree of foresight—a "vision" of the structure of the overall system. Consequently, in the early stages of development, classes may get in the way; a design change may require that the entire class hierarchy of the system be redesigned. Further, classes are not well suited to capturing idiosyncratic behaviour: an object that differs from other objects in a class by some detail of its behavior requires that a new class be built to accommodate it.

On the other hand, as software projects mature, wholesale design changes are less likely to occur, and the rigidity, strong typing, and stability provided by classes become more valuable. For these reasons, object-based languages are often used for prototyping software systems in the early stage of their development (indeed, object-based languages are sometimes referred to as "prototyping" languages), while the final production system might be written in a class-based language. Some languages, such as TypeScript, support both styles for this reason.

Smalltalk is quite strictly class-based, so we will instead use mostly the syntax of *Self* **?** in this section. Self's syntax is similar to, and derived from, Smalltalk's. To make Self a *bit* less foreign, we'll replace some of its syntax with the Smalltalk equivalent.

15.1 Code Sharing in Object-Based Languages

In the absence of classes, we must have a syntax to define objects directly, and field and method declarations must take place within the objects themselves, as in the following Self example:

```
x = (|
    n = 15.
    f = (
        self n := self n + 1
    )
|).
```

This example defines an object with fields `n` and `f`, in which the field `f` is bound to a method. That object is bound to the name `x`. Note that we don't distinguish fields from methods in object-based languages: to put a method on an object, we assign it to a field. As a consequence, methods are values in object-based languages. We can strip a method from its object, assign it to a field of another object, and call it, and `self` will then be the second object. This makes it extremely difficult to define sound type rules for an object-based language. For example, TypeScript allows you to strip methods from their objects (because TypeScript is a type system bolted onto JavaScript, and JavaScript is an object-based language), but TypeScript's method type doesn't contain any information on `self`, so the resulting method stripped of its object is usually unsafe to use. This lack of safety is not caught by TypeScript's type system.

Of particular interest in the study of object-based languages are the mechanisms by which we achieve code sharing in the absence of classes. Clearly, our conceptions about inheritance must change. We will abandon typing entirely; structural typing of object-based languages is possible, but beyond the scope of this course.

Cloning. Most object-based languages allow objects to be *cloned*, i.e., shallowly duplicated. For instance, in Self, `x clone` will create a duplicate of the above `x`, like so:

```
y = x clone.
```

The object `y` is then an exact copy of `x`, having fields `n` and `f`. Using cloning, we may create any number of objects with identical behavior. Since methods are just fields, we can specialize a cloned object's behavior by changing that field in a clone.

Prototyping. While cloning works well, it's fairly restrictive. Self also allows *prototyping*, by which an object may delegate behavior to another object. For instance, we can create the following object:

```
z = (| prototype* = x |).
```

When a field is looked up in `z`, if it's not directly defined in the object, it instead looks for it in the `prototype*` field. If it finds a method in the prototype, in this case `x`, the `self` parameter will still be `z`, so that method can access `z`'s fields. In this way, we can build classes fairly directly with prototypes: we can build all the methods into one prototype, and then clone an object that has all the correct fields. Of course, `prototype*` is also a field, so if we're feeling especially cruel, we can actually *change* an object's prototype, though that's not a very good idea. As a

final twist, Self allows an object to have multiple prototypes—in fact, any field named with a * is a prototype—and thus supports multiple inheritance. It uses priority to disambiguate, with alphabetically earlier names prioritized over alphabetically later names.

The constructor pattern. In Self, it's common to use prototypes and a constructor method, like the following:

```
Rectangle = (|
    withWidth: x height: y = (
        | r |
        r := (|
            prototype* = self.
            width = x.
            height = y
        |).
        ^r
    ).

    area = (
        ^self width * self height
    )
|)
```

The `withWidth:height:` method is meant to be called on `Rectangle` itself, and creates an object with `Rectangle` (as `self`) as a prototype. The `area` method actually won't work if called on `Rectangle` itself, because it will look for fields named `width` and `height`, but no such fields exist on `Rectangle`. Instead, objects created by `withWidth:height:` will have these fields, and through their prototype, the `area` method. A much more popular prototype-based language, JavaScript, does not support cloning, and directly supports the constructor pattern.

16 Miscellany

An object-oriented system usually requires some initial configuration by the implementation, which cannot be implemented in the language itself. For instance, Smalltalk requires every class to be a subclass of some other class, *except* for `Object`. There is no way to create a class without a superclass in Smalltalk, so `Object` must be defined by the implementation. The same is true of blocks, and the `Class` class, as well as the special values `true`, `false`, and `nil`. On the other hand, the classes *for* those special values—`True`, `False`, and `Undefined`, respectively—can actually be defined in Smalltalk, and are.

Because Smalltalk systems are usually environments, rather than files, they have *images*. An image is a file-based representation of the state of all objects in a Smalltalk system—bearing in mind that classes are objects of the `Class` class—which can then be loaded to recreate the environment. Essentially, an image is a file representing σ and Σ. GNU Smalltalk was chosen for this course because it's incredibly difficult to grade an image, for the same reason as it's difficult to define a formal semantics without a starting statement.

Although neither Smalltalk nor Self are themselves particularly popular, their implementations have had profound effects on computing. The concept of Just-in-Time Compilation—compiling code as the program runs—was invented for Smalltalk, refined in Self, and then popularized in Java and JavaScript. This is the usual fate of a research language: its concepts are used, but the language is not. For most researchers, this is the desirable fate of their language, because they get all the credit and none of the maintenance!

17 Fin

In the next module, we will look at *concurrency*, through the lens of Erlang, which makes concurrency feel similar to object orientation.

Rights