Project Ceylon

A Better Java

Version: For internal discussion only

Table of Contents

A work in progress

This project is the work of a team of people who are fans of Java, and of the Java ecosystem, of its practical orientation, of its culture of openness, of its developer community, of its unashamed participation in the world of business computing, and of its strong commitment to portability. However, we recognize that the language and class libraries, designed more than 15 years ago, are no longer the best foundation for a range of today's business computing problems.

The goal of this project is to make a clean break with the legacy Java SE platform, by improving upon the Java language and class libraries, and by providing a modular architecture for a new platform based upon the Java Virtual Machine.

Of course, we recognize that the ability to interoperate with existing Java code, and leverage existing investment in the Java ecosystem, is a critical requirement of any successor to the Java platform.

Java is a simple language to learn and Java code is easy to read and understand. Java provides a level of typesafety that is appropriate for business computing and enables sophisticated tooling with features like refactoring support, code completion and code navigation. Ceylon aims to retain the overall model of Java, while getting rid of some of Java's warts, including: primitive types, arrays, constructors, getters/setters, checked exceptions, raw types, wildcard types, thread synchronization, finalizers, serialization, unsafe typecasts and reflection, and the dreaded `NullPointerException`.

Ceylon has the following goals:

- to be appropriate for large scale development, but to also be *fun*,
- to execute on the JVM, and interoperate with Java code,
- to be easy to learn for Java and C# developers,
- to eliminate some of Java's verbosity, while retaining its readability—Ceylon does *not* aim to be the least verbose/most cryptic language around,
- to provide a declarative syntax for expressing hierarchical information like user interface definition, externalized data, and system configuration, thereby eliminating Java's dependence upon XML,
- to provide a more elegant and more flexible annotation syntax to support frameworks and declarative programming,
- to support and encourage a more functional style of programming with immutable objects and first class functions, alongside the familiar imperative mode,
- to expand compile-time typesafety with compile-time safe handling of null values, compile-time safe typecasts, and a more typesafe approach to reflection,
- to provide language-level modularity,
- to improve on Java's very primitive facilities for meta-programming, thus making it easier to write frameworks, and
- to make it easy to *get things done*.

Unlike other alternative JVM languages, Ceylon aims to completely replace the legacy Java SE class libraries.

Therefore, the Ceylon SDK will provide:

- the OpenJDK virtual machine,
- a compiler that compiles both Ceylon and Java source,
- Eclipse-based tooling,
- a module runtime, and
- a set of class libraries that provides much of the functionality of the Java SE platform, together with the core functionality of the Java EE platform.

Chapter 1. Introduction

Ceylon is a statically-typed, general-purpose, object-oriented language featuring a syntax similar to Java and C#. Ceylon programs execute in any standard Java Virtual Machine and, like Java, take advantage of the memory management and concurrency features of that environment. The Ceylon compiler is able to compile Ceylon code that calls Java classes or interfaces, and Java code that calls Ceylon classes or interfaces. Ceylon improves upon the Java language and type system to reduce verbosity and increase typesafety compared to Java and C#. Ceylon encourages a more functional style of programming, resulting in code which is easier to reason about, and easier to refactor. Moreover, Ceylon provides its own native SDK as a replacement for the Java platform class libraries.

1.1. Language overview

1.1.1. The type system

Ceylon features a similar inheritance and generic type model to Java. A type is either an *interface* or a *class*. An interface may extend an arbitrary number of other interfaces. A class may implement an arbitrary number of interfaces and must extend another class.

There are no primitive types or arrays in Ceylon, so all values are instances of the type hierarchy root `lang.Object`. However, the Ceylon compiler is permitted to optimize certain code to take advantage of the better performance of primitive types on the JVM.

Ceylon does not support Java-style wildcard type parameters or raw types. Instead, like Scala, a type parameter may be marked as covariant or contravariant by the class or interface that declares the parameter. Type arguments are reified in Ceylon, eliminating many problems related to type erasure in Java.

Ceylon supports *type aliases*, similar to C-style `typedef`.

1.1.2. Compiler-enforced naming conventions

The Ceylon compiler enforces the traditional Smalltalk naming convention: type names begin with an initial uppercase letter—for example, `Liberty` or `RedWine`—member names and local names with an initial lowercase letter or underscore—for example, `blonde`, `immanentize()` or `boldlyGo()`. This innovation allows a much cleaner syntax for program element annotations than the syntax found in either Java or C#.

1.1.3. Class initialization and instantiation

Ceylon does not feature any Java-like constructor declaration and so each Ceylon class has a formal parameter list, and exactly one *initializer*—the body of the class. This helps reduce verbosity and results in a more regular block structure.

In place of constructor overloading, Ceylon allows class names to be overloaded. Even better, member classes of a class may be overridden by subclasses. Instantiation is therefore a polymorphic operation in Ceylon, eliminating the need for factory methods.

1.1.4. Methods and attributes

Ceylon types have members: *methods* and *attributes*. Ceylon methods are similar to Java methods. However, Ceylon classes do not contain fields, in the traditional sense. Instead, Ceylon supports only a higher-level construct: polymorphic attributes, which are similar to C# properties. Attributes abstract the internal representation of the state of an object.

There are no `static` members. Instead, *toplevel* methods are declared as direct members of a package. This, along with certain other features, gives the language a more regular block structure.

1.1.5. Defaulted parameters

As an alternative to method or class overloading, Ceylon supports method and class initialization parameters with default values.

1.1.6. First-class functions

Ceylon supports first-class function types and higher-order functions, with minimal extensions to the traditional C syntax. A method declaration may specify a *callable parameter* that accepts references to other methods with a certain signature. The argument of such a callable parameter may be either a reference to a named method declared elsewhere, or a new method defined inline as part of the method invocation. A method may even return an invocable reference to another method. Finally, nested method declarations receive a closure of immutable values in the surrounding scope.

1.1.7. Immutability by default

By default, Ceylon classes, attributes and locals are immutable. Mutable classes, attributes and locals must be explicitly declared using the `mutable` annotation. An immutable class may not declare `mutable` attributes or extend a `mutable` class. An immutable attribute or local may not be assigned after its initial value is specified.

1.1.8. Compile-time safety for optional values and type narrowing

By default, Ceylon attributes and locals do not accept null values. Optional locals and attributes must be explicitly declared. Optional expressions are not assignable to non-optional locals or attributes, except via use of the `if (exists ... )` construct. Thus, the Ceylon compiler is able to detect illegal use of a null value at compile time. Therefore, there is no equivalent to Java's `NullPointerException` in Ceylon.

Ceylon does not feature C-style typecasts. Instead, the `if (is ... )` and `case (is ... )` constructs may be used to narrow the type of an object reference without risk of a `ClassCastException`.

1.1.9. Control flow

Ceylon control flow structures are very similar to the traditional constructs found in C, C# and Java. However, inline methods can be used together with a special Smalltalk-style method invocation protocol to achieve more specialized flow control and other more functional-style constructs such as comprehensions.

Ceylon features an exceptions model inspired by Java and C#. Checked exceptions are not supported.

1.1.10. Operators

Ceylon features a rich set of operators, including most of the operators supported by C and Java. True operator overloading is not supported. However, each operator is defined to act upon a certain class or interface type, allowing application of the operator to any class which extends or implements that type. This is called *operator polymorphism*.

1.1.11. Numeric types

Ceylon's numeric type system is much simpler than C, C# or Java, with exactly five built-in numeric types (compared to eight in Java and eleven in C#). The built-in types are classes representing natural numbers, integers, floating point numbers, arbitrary precision integers and arbitrary precision decimals. `Natural`, `Integer` and `Float` values are 64 bit by default, and may be optimized for 32 bit architectures via use of the `small` annotation.

1.1.12. Extensions

True open classes are not supported. However, Ceylon supports *extensions*, which allow addition of methods and interfaces to existing types, and transparent conversion between types, within a textual scope. Extensions only affect the operations provided by a type, not its state.

1.1.13. Structured data

Ceylon introduces a set of syntax extensions that support the definition of domain-specific languages and expression of structured data. These extensions include specialized syntax for initializing objects and collections and expressing literal values or user-defined types. The goal of this facility is to replace the use of XML for expressing hierarchical structures such as documents, user interfaces, configuration and serialized data. An especially important application of this facility is Ceylon's built-in support for program element annotations.

1.1.14. Metaprogramming

Ceylon provides sophisticated support for meta-programming, including a typesafe metamodel and events. This facility is inspired by similar features found in dynamic languages such as Smalltalk, Python and Ruby, and by the much more complex features found in aspect oriented languages like Aspect J. Ceylon does not, however, support aspect oriented programming as a language feature.

1.1.15. Modularity

Ceylon features language-level *package* and *module* constructs, and language-level access control with five levels of visibility for program elements: block local (the default), `private`, `package`, `module` and `public`. There's no equivalent to Java's `protected`.

1.2. A brief tutorial

1.2.1. Writing a simple program in Ceylon

Here's a classic example, implemented in Ceylon:

```
doc "The classic Hello World program"
public void hello(Process process) {
    process.writeLine("Hello, World!");
}
```

This code defines a method named `hello()` with a parameter of type `lang.Process`. The method is annotated `public`, which makes it accessible to code in other compilation units. When the method is executed, it calls the `writeLine()` method of the class `Process`. (This method displays its parameter on the console.)

The `doc` annotation contains documentation that is included in the output of the Ceylon documentation compiler.

Copy this code into a file named `hello.ceylon` and put the file in the `ceylon/source/` directory of your Ceylon SDK installation.

Now, in the `ceylon` directory, run the following command:

```
ceylon hello
```

This command compiles the source to bytecode and runs the Java virtual machine. You should see the following output in the console:

```
Hello World!
```

1.2.2. Dealing with objects that aren't there

This improved version of the program takes a name as input from the command line. We have to account for the case where nothing was specified at the command line, which gives us an opportunity to explore how null values are treated in Ceylon, which is quite different to what you're probably used to in Java or C#.

```
doc "Print a personalized greeting"
public void hello(Process process) {
    String? name = process.args.first;
    String greeting;
    if (exists name) {
        greeting = "Hello, " name "!";
    }
    else {
        greeting = "Hello, World!";
    }
    process.writeLine(greeting);
}
```

The `Process` class has an attribute named `args`, which holds a `List` of the program's command line arguments. The local `name` is initialized with the first of these arguments, if any. This local is declared to have type `String?`, to indicate that it may contain a null value. The `if (exists ...)` control structure is used to initialize the value of the non-null local named

`greeting`, interpolating the value of `name` into the message string whenever `name` is not null. Finally, the message is printed to the console.

Unlike Java, locals, parameters, and attributes that may contain null values must be explicitly declared as being of type `Optional<X>` where `X` is the type of value they contain when not holding a null value. Ceylon lets us abbreviate the type name `Optional<X>` to `X?`. The value `null` is an instance of type `Optional<X>`, but it's not an instance of `Object`. So there's simply no way to assign `null` to a local that isn't of type `Optional`. The compiler won't let you.

Nor will the Ceylon compiler let you do anything "dangerous" with a value of type `Optional`—that is, anything that could cause a `NullPointerException` in Java—without first checking that the value is not null using `if (exists ... )`.

If you're worried about the performance implications of wrapping values in instances of `Optional`, don't be. `Optional` isn't a *reified* type—it exists at compile time, when the compiler is validating your code for typesafety, but then the compiler eliminates it as part of the compilation process, allowing the resulting bytecode to take advantage of the efficient handling of null values in the virtual machine.

It's possible to declare the local `name` inside the `if (exists ... )` condition:

```
String greeting;
if (exists String name = process.args.first) {
    greeting = "Hello, " name "!";
}
else {
    greeting = "Hello, World!";
}
process.writeLine(greeting);
```

This is the prefered style most of the time, since we can't actually use `name` for anything useful outside of the `if (exists ... )` construct.

Copy this new version of the program into `hello.ceylon` and run, as before:

```
ceylon hello
```

You should see the same output as before:

```
Hello, World!
```

Now, run the program again, with a command-line argument:

```
ceylon hello everybody
```

You should see the following output:

```
Hello everybody!
```

1.2.3. Creating your own classes

Our method now has too many responsibilities, and is not at all reusable. Let's refactor the code. Ceylon is an object oriented language, so we usually write most of our code in *classes*. A class is a type that packages:

- operations—called *methods*,
- state—held by *attributes*, and,
- sometimes, other nested types.

Types (interfaces, classes, and aliases) have names that begin with uppercase letters. Members (methods and attributes) and locals have names that begin with lowercase letters. This is the rule you're used to from Java. Unlike Java, the Ceylon compiler enforces these rules. If you try to write `class hello` or `String Name`, you'll get a compilation error.

Just like in Java or C#, a class defines the accessibility of its members using *visibility modifier annotations*, allowing the class to hide its internal implementation from clients. Unlike Java, members are hidden from code outside the body of the class *by default*—only members with explicit visibility modifiers are visible to other toplevel types or methods, other compilation units, other packages, or other modules.

Our first version of the `Hello` class has a single attribute and a single method, both declared to have `package` visibility, making them accessible to other code in the same package:

```
doc "A personalized greeting"
class Hello(String? name) {

    doc "The greeting"
    package String greeting;
    if (exists name) {
        greeting = "Hello, " name "!";
    }
    else {
        greeting = "Hello, World!";
    }

    doc "Print the greeting"
    package void say(OutputStream stream) {
        stream.writeLine(greeting);
    }

}
```

To understand this code completely, we're going to need to first explore the concept of an attribute, and then discuss how object initialization works in Ceylon.

1.2.4. Abstracting state using attributes

The attribute `greeting` is a *simple attribute*, very similar to a Java field. Its value is specified immediately after it is declared. Usually we can declare and specify the value of an attribute in a single line of code.

```
package String greeting = "Hello, " name "!";
```

An attribute is a bit different to a Java field. It's an abstraction of the notion of a value. Some attributes are simple value holders like the one we've just seen; others are more like a getter method, or, sometimes, like a getter and setter method pair. Like methods, attributes are polymorphic—an attribute definition may be overridden by a subclass.

We could rewrite the attribute `greeting` as a *getter*:

```
package String greeting {
    if (exists name) {
        return "Hello, " name "!"
    }
    else {
        return "Hello, World!"
    }
}
```

Clients of a class never need to know whether the attribute they access holds state directly, or is a getter that derives its value from other attributes of the same object or other objects. In Ceylon, you don't need to go around declaring all your attributes `private` and wrapping them in getter and setter methods. Get out of that habit right now!

1.2.5. Understanding object initialization

In Ceylon, classes don't have constructors. Instead:

- the parameters needed to instantiate the class—the *initialization parameters*—are declared directly after the name of the class, and
- code to initialize the new instance of the class—the *class initializer*—goes directly in the body of the class.

Take a close look at the following code fragment:

```
String greeting;
if (exists name) {
    greeting = "Hello, " name "!";
}
else {
    greeting = "Hello, World!";
}
```

In Ceylon, this code could appear in the body of a class, where it would be declaring and specifying the value of an immutable attribute, or it could appear in the body of a method definition, where it would be declaring and specifying the value of an immutable local variable. That's not the case in Java, where initialization of fields looks very different to initialization of local variables! Thus the syntax of Ceylon is more *regular* than Java. Regularity makes a language easy to learn and easy to refactor.

Now let's turn our attention to a different possible implementation of `greeting`:

```
class Hello(String? name) {

    package String greeting {
        if (exists name) {
            return "Hello, " name "!"
        }
        else {
            return "Hello, World!"
        }
    }

    ...

}
```

You might be wondering why we're allowed to use the parameter `name` inside the body of the getter of `greeting`. Doesn't the parameter go out of scope as soon as the initializer terminates? Well, that's true, but Ceylon is a language with a very strict block structure, and the scope of declarations is governed by that block structure. In this case, the scope of `name` is the whole body of the class, and the definition of `greeting` sits inside that scope, so `greeting` is permitted to access `name`.

We've just met our first example of *closure*, a concept from functional programming. We say that method and attribute definitions receive a closure of immutable values defined in the class body to which they belong. That's just a fancy way of obfuscating the idea that `greeting` holds onto the value of `name`, even after the initializer completes.

1.2.6. Instantiating classes and overloading their initialization parameters

Oops, I got so excited about attributes and closure that I forgot to show you the code that uses `Hello`!

```
doc "Print a personalized greeting"
public void hello(Process process) {
    Hello(process.args.first).say(process);
}
```

Our rewritten `hello()` method just creates a new instance of `Hello`, and invokes `say()`. Ceylon doesn't need a `new` keyword to know when you're instantiating a class. No, we don't know why Java needs it. You'll have to ask James.

I suppose you're worried that if Ceylon classes don't have constructors, then they also can't have multiple constructors. Does that mean we can't overload the initialization parameter list of a class? Well, not exactly. Ceylon doesn't have constructor overloading, but it does have *class overloading*. Believe it or not, Ceylon lets you write multiple classes with the same name! Let's overload `Hello`:

```
doc "A command line greeting"
class Hello(Process process)
    extends Hello(process.args.first) {}
```

A class can overload a second class by extending it and declaring different initialization parameters. An overloaded class with an empty body is the Ceylon approach to "constructor" overloading. Of course, overloaded classes can do much more than just this!

Our `hello()` method is now looking *really* simple:

```
doc "Print a personalized greeting"
public void hello(Process process) {
    Hello(process).say(process);
}
```

1.2.7. Inheritance and overriding

In object-oriented programming, we often replace conditionals (`if`, and especially `switch`) with subtyping. Let's try refact-

oring `Hello` into two classes, with two different implementations of `greeting`:

```
doc "A default greeting"
class DefaultHello() {

    doc "The greeting"
    package default String greeting = "Hello, World!";

    doc "Print the greeting"
    package void say(OutputStream stream) {
        stream.writeLine(greeting);
    }

}
```

Notice that Ceylon forces as to declare attributes or methods that can be overridden by annotating them `default`.

Subclasses specify their superclass using the `extends` keyword, followed by the name of the superclass, followed by a list of arguments to be sent to the superclass initialization parameters. It looks just like an expression that instantiates the superclass:

```
doc "A personalized greeting"
class PersonalizedHello() extends DefaultHello() {

    doc "The personalized greeting"
    override String greeting {
        return "Hello, " name "!"
    }

}
```

Ceylon also forces us to declare that an attribute or method overrides an attribute or method of a superclass by annotating it `override`. All this annotating stuff costs a few extra keystrokes, but it helps the compiler detect errors. We can't inadvertantly override a member or the superclass, or inadvertantly *fail* to override it.

On the other hand, we don't need to declare the visibility of the attribute annotated `override`. The `package` annotation is inherited from the attribute it overrides.

There's one problem with what we've just seen. A personalized greeting is not really a kind of default greeting. This is a case for introducing an abstract superclass:

```
doc "A greeting"
abstract class Hello() {

    doc "The (abstract) greeting"
    package String greeting;

    doc "Print the greeting"
    package void say(OutputStream stream) {
        stream.writeLine(greeting);
    }

}
```

Ceylon requires us to annotate abstract classes `abstract`, just like Java. This annotation specifies that a class cannot be instantiated, and can define abstract members. Unlike Java, Ceylon doesn't require us to annotate abstract members—simply leaving out the implementation of the member is perfectly sufficient for the compiler to realize that it is abstract. An attribute which is never initialized is an abstract attribute. Ceylon doesn't have default attribute values like Java does.

One way to define an implementation for an inherited abstract attribute is to simply assign a value to it in the subclass.

```
doc "A default greeting"
class DefaultHello() extends Hello() {

    greeting = "Hello, World!";

}
```

Of course, we can also define an implementation for an inherited abstract attribute by overriding it.

```
doc "A personalized greeting"
class PersonalizedHello() extends Hello() {

    doc "The personalized greeting"
    override String greeting {
```

```
        return "Hello, " name "!"
    }

}
```

1.2.8. Working with mutable state

Ceylon encourages you to use *immutable* attributes as much as possible. An immutable attribute has its value specified when the object is initialized, and is never reassigned. If we want to be able to assign a value to a simple attribute we need to annotate it `mutable`:

```
package mutable String greeting := "Hello World";
if (exists name) {
    greeting := "Hello, " name "!";
}
```

Notice the use of `:=` instead of `=` here. This is important! In Ceylon, specification of an immutable value is done using `=`. Assignment to a `mutable` attribute or local is considered a different kind of thing, always performed using the `:=` operator.

To force you to think twice before adding a `mutable` attribute to your class, Ceylon requires that classes with mutable attributes be explicitly annotated `mutable`. Hopefully, you'll find yourself doing this less frequently than you would do it in Java.

If we want to make an attribute with a getter mutable, we need to define a matching *setter*. Usually this is only useful if you have some other internal attribute you're trying to set the value of indirectly.

Suppose our class has the following mutable simple attribute, intended for internal consumption only:

```
mutable String grtng := "Hello World";
```

Then we can abstract the simple attribute using a mutable attribute defined as a getter/setter pair:

```
doc "gets the greeting"
public String greeting {
    return grtng
}

doc "sets the greeting"
package assign greeting {
    grtng = greeting;
}
```

Yes, this is a *lot* like a Java get/set method pair. But since Ceylon attributes are mutable, and since you can redefine a simple attribute as a getter or getter/setter pair without affecting clients that call the attribute, you rarely need to write this code unless you're doing something special in the getter or setter. So here's a more realistic example:

```
doc "gets the greeting"
public String greeting {
    return grtng
}

doc "sets the greeting"
package assign greeting {
    if (empty greeting) {
        throw IllegalArgumentException("greeting can not be empty)
    }
    else {
        grtng = greeting.strip().normalize();
    }
}
```

Get used to using `empty` to check that a value of type `String` or `String?` is an actual non-empty string of characters. It's much more readable than the equivalent Java idiom `!"".equals(greeting)`.

1.2.9. Using numeric types

Ceylon doesn't have anything like Java's primitive types. The types that represent numeric values are just ordinary classes. Ceylon has fewer numeric types than other C-like languages:

- `Natural` represents the unsigned integers and zero,
- `Integer` represents signed integers,
- `Float` represents floating point decimal numbers,
- `Whole` represents arbitrary-precision signed integers, and
- `Decimal` represents arbitrary-precision and arbitrary-scale decimals.

`Natural`, `Integer` and `Float` have 64-bit precision by default. You can specify that a value has 32-bit precision by annotating it `small`.

There are only two kinds of numeric literals: literals for `Naturals`, and literals for `Floats`:

```
Natural one = 1;
```

```
Float oneHundredth = 0.01;
```

```
Float oneMillion = 1.0E+6;
```

Widening type conversions are implicit, since the `lang` package defines a set of built-in *extensions* (a concept we'll meet later) that take care of this:

```
Decimal oneTenth = 0.1;
```

```
Whole zero = 0;
```

```
Integer minusOne = -1;
```

To perform a narrowing type conversion, you need to call one of the operations (well, they're attributes, actually) defined by the interface `Number`:

```
Natural one = 1.003.natural;
```

```
Integer int = whole.integer;
```

Of course, a narrowing conversion can result in an exception at run time, so take care!

You can use all the operators you're used to from other C-style languages with the numeric types. You can also use the `**` operator to raise a number to a power:

```
Float diagonal = sqrt(length**2+width**2);
```

Of course, if you want to use the increment `++` operator, decrement `--` operator, or one of the compound assignment operators such as `+=`, you'll have to declare the value `mutable`.

```
module class Counter() {

    mutable Natural count := 0;

    module Natural next() {
        return ++count
    }

}
```

You can even apply the bitwise and, or, xor, and complement operators to the type `Natural`:

```
Natural result = color&mask;
```

Operators in Ceylon are, in principle, just abbreviations for some expression involving a method call. So the numeric types all implement the `Numeric` interface, which declares the methods `plus()`, `minus()`, `times()`, `divided()` and `power()`. The numeric operators are defined in terms of these methods of `Numeric`. And `Natural` gets its bitwise operations from the

`Bits` interface.

But don't worry about the performance implications of this—in practice, the compiler is permitted to optimize away the method invocations. In fact, the compiler is permitted to optimize the built-in numeric types down to the virtual machine's native numeric types.

The real value of `Numeric` is that you can implement the interface yourself, to introduce your own, more specialized numeric type. And you'll be able to apply all the usual numeric operators to it. This is what we call *operator polymorphism*.

1.2.10. A quick overview of collections

The collections package is one of the best features of Ceylon. Let's briefly meet some of the main characters.

There's no arrays in Ceylon, but you can write the following code:

```
String[] voyage = { "Melbourne", "San Francisco", "Atlanta", "Guanajuato" };
String? sf = voyage[2];
String[] usLeg = voyage[1..2];
String[] longerVoyage := voyage + { "Rome", "Paris", "Edinburgh" };
```

The syntax `String[]` is just an abbreviation for the type `Sequence<String>`. Don't be scared by the fancy syntax—it's just sugar. There's actually nothing much special about the interface `Sequence` from the point of view of the type system. But sequences of values are a common occurrence in computing, so Ceylon provides a streamlined syntax for dealing with them.

Just like in Java, the collection types are *parameterized* or *generic* types. Unlike Java, there are no raw types—if a type declares type parameters, a type argument for each parameter is required everywhere the type is used. For example, the following declaration is not legal:

```
Sequence voyage = { "Atlanta", "Boston", "San Francisco" };  //error! missing type argument
```

The closest thing to a raw type, in this case, would be a `Sequence<Object>`.

```
Sequence<Object> voyage = { "Atlanta", "Boston", "San Francisco" };
```

Which we could abbreviate, of course:

```
Object[] voyage = { "Atlanta", "Boston", "San Francisco" };
```

Sequences of ordinal values are especially common, so there's an operator for constructing them, called the *range* operator:

```
Natural[] from1To10 = 1..10;
```

```
Integer[] fromNegative10To10 = -10..10;
```

```
Character[] fromAToZ = `A`..`Z`;
```

The `Sequence` interface as assignable to `Iterable`, so we can iterate a `Sequence` using a `for` loop:

```
for (String city in voyage) {
    stream.writeLine(city);
}
```

If, for some reason, we need access to the index of each element, we can also do the following:

```
for (Natural i -> String city in voyage) {
    stream.writeLine($i + ": " + city);
}
```

This works because there's a built-in extension method (a concept we'll meet later) that produces an `Iterable<Entry<Natural,X>>` for any `Sequence<X>`. (Oh, the `$` operator just converts an object to a printable string.)

Sometimes the keys of our collection elements aren't natural numbers, so we need something other than a `Sequence`. A `Correspondence<U,V>` is a mapping from keys of type `U` to values of type `V`.

```
Correspondence<String,String> cities = { "Emmanuel"->"Paris",
                                         "Andrew"->"Cambridge",
                                          "Pete"->"Edinburgh",
                                         "Gavin"->"Guanajuato" };
```

In fact, a `Sequence` is just a special case of a `Correspondence` where the keys are `Natural` numbers. `Sequence<X>` is a subtype of `Correspondence<Natural, X>`.

We can extract values from a `Correspondence` by key using the lookup operator:

```
String? paris = cities["Emmanuel"];
```

Notice that the lookup operator returns an `Optional` result, to account for the possibility that there is no value defined for the given key.

So far we haven't met anything from the collections package proper. `Sequence` and `Correspondence` are part of the package `lang`. They're there to abstract away the nice syntax we've just met from the (relatively) gory details of the collections package.

Inside the collections package you'll find some interfaces with pretty familiar names: `Collection`, `Set`, `List` and `Map`. (There's even a `Bag`.) But these interfaces aren't quite what you're used to from Java. Most importantly, none of them provide operations to change the elements of the collection. If you need to mutate a collection, you'll need one of their evil twins: `OpenCollection`, `OpenSet`, `OpenList` and `OpenMap` (and we can't forget `OpenBag`).

A `Map` is the most important kind of `Correspondence`. We can add or override entries in an `OpenMap` using the assignment operator:

```
OpenMap<String,String> cities = {}
cities["Gavin"] := "Guanajuato";
cities["Emmanuel"] := "Paris";
cities["Pete"] := "Edinburgh";
cities["Andrew"] := "Cambridge";
```

It's even possible to define multiple entries at once:

```
OpenMap<String,String> cities = {}
cities.define("Emmanuel"->"Paris",
                "Gavin"->"Guanajuato",
                 "Pete"->"Edinburgh",
               "Andrew"->"Cambridge");
```

The `->` operator constructs an `Entry` from its operands. A `Map` is a collection of entries, a `Collection<Entry>`. This is a big improvement over Java where, unaccountably, a `Map` isn't actually a `Collection`!

Since `Collections` are `Iterable`, we can iterate a `Map` like this:

```
for (Entry<String,String> entry in cities) {
    stream.writeLine(entry.key + " lives in " + entry.value);
}
```

Or, much more commonly:

```
for (String person -> String city in cities) {
    stream.writeLine(name + " lives in " + city);
}
```

(Note that in this case, the `->` symbol isn't really the entry construction operator we saw above. In this case it's actually part of the syntax of the `for` loop.)

A `List` is the most important kind of `Sequence`. We usually add entries to an `OpenList` using the `append()` method:

```
OpenList<String> voyage = {}
voyage.append("Melbourne");
voyage.append("San Francisco");
voyage.append("Atlanta);
```

It's even possible to add multiple elements at once:

```
OpenList<String> voyage = {}
voyage.append("Melbourne", "San Francisco", "Atlanta");
```

We can also change the value at a particular index in an `OpenList`:

```
voyage[1] := "Los Angeles";
```

Splitting the collection interfaces into a "read-only" view and a "writable" view has several advantages, but the most important is that you now don't need to worry about exposing collections as part of the public contract of your classes. In Java, it's always difficult for a client to know if you're returning a copy of your internal list, a reference to your internal list (which you might have done inadvertantly), or a reference wrapped in an `immutableList()`—you know, that wonderful beastie which helpfully throws exceptions at runtime to let you know that it is supposed to be "read-only". In Ceylon, you can protect the state of your internal data structures by simply not returning the "writable" view to clients. Send them back a `List`, and they know what they're allowed to do with it.

1.2.11. Taking advantage of functional-style programming

Let's go back, once again, to the attribute `greeting` of `Hello`:

```
package String greeting {
    if (exists name) {
        return "Hello, " name "!"
    }
    else {
        return "Hello, World!"
    }
}
```

This definition works well enough, but it's quite procedural, with two `return` statements. That's not usually considered good style in Ceylon, though there are certainly times when it's necessary. Instead, Ceylon lets you write code like this using a more functional style.

In procedural programming we usually pass a list of values to a method, which performs computations using those values, and returns another value. In functional programming, we can pass an operation to a method, which calls our operation, and returns a value, or, perhaps, a different operation.

For example, the Ceylon standard librares define a method called `ifExists()` that accepts an object and two *callable objects*. (A method parameter that accepts a callable objects is called a *callable parameter*.) If the object is not null, `ifExists()` invokes the first method. Otherwise, `ifExists()` invokes the second method. We can use `ifExists()` to rewrite `greeting`:

```
package String greeting {
    String helloName() { return "Hello, " name "!" }
    String helloWorld() { return "Hello, World!" }
    return ifExists(name, helloName, helloWorld)
}
```

Well, that's certainly more functional, but it's also a lot more verbose. But this is not the way `ifExists()` is really intended to be used. Ceylon provides a special method argument syntax for defining a method that is passed to another method inline, as part of the invocation. An inline method definition is called an *inline callable argument*. Inline callable arguments follow the normal parenthesized argument list.

As a first step, we could rewrite `greeting` as follows, to make it really clear that we're defining methods that are passed as arguments to the callable parameters `then` and `otherwise`.

```
package String greeting {
    return ifExists(name)
        //first inline callable argument
        then () {
            return "Hello, " name "!"
        }
        //second inline callable argument
        otherwise () {
            return "Hello, World!"
        }
}
```

Note that there are three `return` statements here. The nested `return` statements specify the return values of the two inline

methods. They do not end the execution of `greeting`!

This syntax was chosen to make it possible to define new control structures that closely mimic the syntactic form of the traditional C-style built-in control structures.

Usually, we would abbreviate the above code by:

- eliminating the empty formal parameter lists—we can leave off the `()`, and
- specifying the method return values in parentheses, instead of writing `return` statements surrounded by braces—we're allowed to write `("Hello, World!")` instead of `{ return "Hello, World!" }`.

The abbreviated code looks like this:

```
package String greeting {
    return ifExists(name)
        then ("Hello, " name "!")
        otherwise ("Hello, World!")
}
```

This syntax might look a little unfamiliar at first, but you'll soon get used to it. In Ceylon, we use method invocations with inline callable arguments to express things that are difficult to express without specialized syntax in other languages, for example:

- *assertion*, such as: `assert() that (x==0.0);`
- *comprehension*, such as: `String[] names = from (people) select (Person p) (p.name);`
- *quantification*, such as: `Boolean adults = forAll (people) every (Person p) (p.age>=18);`
- *repetition*, such as: `repeat (3) times { stream.writeLine("Hello!"); }`

Look at each of these examples, and ask yourself:

- What is the name of the method that is being called?
- Are there any ordinary arguments? Which are they?
- Which is the inline callable argument? What is its name?
- Are there any formal parameters of the inline callable argument? What are the parameter names?
- Where is the implementation of the inline callable argument? What happens when the inline callable argument is invoked?

Well, if you thought you were starting to feel comfortable with this new syntax, I've got something extra to throw at you! A method parameter can be declared as an *iterator* method, which allows us to move the declaration of the formal parameters of the inline callable arguments. For example, the `lang` package defines a `from()` method that is intended to be called like this:

```
String[] names = from (Person p in people) select (p.name) where (p.age>18);
```

Which is exactly equivalent to, but much more readable than, the following:

```
String[] names = from (people) select (Person p) (p.name) where (Person p) (p.age>18);
```

We won't go into the details how to declare an iterator method here (it's just a couple of annotations). It's much more important to be able to *use* iterator methods than actually write them yourself.

Oh, by the way, there's an even easier way to define `greeting`, using the `?` operator.

```
package String greeting {
    return "Hello, " (name ? "World") "!"
}
```

1.2.12. Defining user interfaces declaratively

Finally, lets create a web-based user interface for our program.

We've seen lots of examples of invoking a method or instantiating a class using the traditional C-style argument list where arguments are surrounded in parentheses and separated by commas. Arguments are matched to parameters by their position in the list. Using this syntax, we could create a tree of objects as follows:

```
Html(
    Head('hello.css', "Hello World"),
    Body(
        Div("greeting", "Hello World"),
        Div("footer", "Powered by Ceylon")
    )
)
```

However, Ceylon provides an alternative way of writing argument lists that makes it much more natural to define heirarchical structures. A *named argument list* is a list of arguments surrounded by braces and separated by semicolons. Varargs parameters in a named argument list don't need names, and are separated by commas. For example:

```
Html {
    head = Head {
        title = "Hello World";
        cssStyleSheet = 'hello.css';
    };
    body = Body {
        Div {
            cssClass = "greeting";
            "Hello World"
        },
        Div {
            cssClass = "footer";
            "Powered by Ceylon"
        }
    };
}
```

We're going to use this syntax to define our web page.

```
import html.*;

doc "A web page that displays a greeting"
page '/hello.html'
public class HelloHtml(Request request)
        extends Html(request) {

    Hello hello = Hello(request.parameters["name"]);

    head = Head {
        title = "Hello World";
        cssStyleSheet = 'hello.css';
    };
    body = Body {
        Div {
            cssClass = "greeting";
            hello.greeting
        },
        Div {
            cssClass = "footer";
            "Powered by Ceylon"
        }
    };

}
```

This program demonstrates Ceylon's support for defining structured data, in this case, HTML. The `HelloHtml` class extends Ceylon's `Html` class and specified its abstract `head` and `body` attributes. The `page` annotation specifies the URL at which this HTML should be accessible. A single-quoted string literal is used, allowing the format of the URL to be validated by the Ceylon compiler.

1.2.13. Defining structured data formats

Let's try and define our own structured data format, to personalize our program a little. First, we'll create a class to represent people:

```
doc "Represents a person"
package class Person(String firstName, String lastName, Language lang) {
    package String firstName = firstName;
    package String lastName = lastName;
    package Language lang = lang;
}
```

Now, we'll create a class with an *enumerated list of instances*—almost exactly like a Java `enum`—to represent the built-in languages that our program supports.

```
doc "Represents a language"
package class Language(String hello) {

    english("Hello"),
    french("Bonjour"),
    spanish("Hola"),
    italian("Ciao")
    ...

    doc "The word for \"hello\" in the language"
    package String hello = hello;

}
```

The use of `...` at the end of the list of enumerated instances means that it's possible to create other `Languages`. Very important in this case, since our list of languages is definitely not exhaustive! If our list of instances were exhaustive, we would terminate it using `;`, thereby preventing other code from instantiating additional `Languages`.

Now we can define the set of people known to our application like this:

```
Set<Person> people = {
        Person {
            firstName = "Gavin";
            lastName = "King";
            language = Language.english;
        },
        Person {
            firstName = "Emmanuel";
            lastName = "Bernard";
            lang = Language.french;
        },
        Person {
            firstName = "Pete";
            lastName = "Muir
            lang = Language.english;
        },
        Person {
            firstName = "Andrew";
            lastName = "Haley";
            lang = Language.english;
        }
    };
```

But where should we put our list of people? Well, just for fun, let's specify them using an annotation of `Hello`. First we'll need to learn how to define new annotations.

1.2.14. Defining annotations

In Ceylon, an annotation is just a method that produces an ordinary type. An annotation for specifying a set of people could be defined like this:

```
doc "An annotation for specifying a list of
     people"
annotation {
    of = classes;
    withType = Greeting;
    occurs = onceEachType;
}
package Set<People> people(Person... people) {
    return HashSet(people);
}
```

The meta-annotation `annotation` specifies that this annotation can only occur an the class `Greeting`, and can only occur once.

Ceylon provides a typesafe *metamodel* for types, classes, interfaces, methods and attributes. This is Ceylon's version of Java's reflection API, but it's much, much easier to use and much more powerful. We'll need to use the metamodel to gain access to program element annotations at runtime.

We're not going to go into all the details of the metamodel here. All you need to know is that in the code we're about to see, the expression `Greeting` evaluates to the metamodel object of type `Class<Greeting>`. We could write:

```
Class<Greeting> greetingClass = Greeting;
```

Similarly, the expression `Set<Person>` evaluates to the metamodel object of type `Type<Set<Person>>`. We could write:

```
Type<Set<Person>> personSetType = Set<Person>;
```

We obtain the annotations of a program element—in this case, the class `Greeting`—using reflection upon the metamodel object that represents the program element. We must specify the type returned by the annotation—in this case, `Set<People>`—by passing its metamodel object.

```
Set<Person>[] annotations = Greeting.annotations(Set<Person>);
```

Careful, there may be multiple annotations producing the same type of object!

We're already ready to use our new annotation. Just like with other method invocations, we have the choice between specifying the arguments of an annotation using a positional parameter list surrounded by parenthesis, or using a named parameter list surrounded by braces. In this case, the braces look more visually appealing:

```
doc "A personalized greeting"
people {
    Person {
        firstName = "Gavin";
        lastName = "King";
        language = Language.english;
    },
    Person {
        firstName = "Emmanuel";
        lastName = "Bernard";
        lang = Language.french;
    },
    Person {
        firstName = "Pete";
        lastName = "Muir
        lang = Language.english;
    },
    Person {
        firstName = "Andrew";
        lastName = "Haley";
        lang = Language.english;
    }
}
class Hello(String? name) {

    doc "The list of people"
    Set<Person> people;
    if (exists Set<Person> annotation =
            type.annotations(Set<Person>).first) {
        people = annotation;
    }
    else {
        throw Exception("Not annotated people")
    }

    doc "The greeting"
    package String greeting {
        if (exists name) {
            if (exists Person person =
                    first (Person p in people)
                        where (p.firstName==name)) {
                return "" person.lang.hello ", "
                    person.firstName " " person.lastName "!"
            }
            else {
                return "Hello, " name "!";
            }
        }
        else {
            return "Hello, World!";
        }
    }
```

```
    doc "Print the greeting"
    package void say(OutputStream stream) {
        stream.writeLine(greeting);
    }

}
```

1.2.15. Generic types and covariance

Programming with generic types is one of the most difficult parts of Java. That's still true, to some extent, in Ceylon. But because the Ceylon language and SDK were designed for generics from the ground up, Ceylon is able to eliminate one of the bits of Java generics that's really hard to get your head around: wildcard types. Wildcard types were Java's solution to the problem of *covariance* in a generic type system. Let's first explore the idea of covariance, and then see how Ceylon's solution, copied from Scala, works.

It all starts with the intuitive expectation that a collection of `Geek`s is a collection of `Person`s. That's a reasonable intuition, but in procedural languages, where collections can be mutable, it turns out to be incorrect. Consider the following possible definition of `Collection`:

```
interface Collection<X> {
    Iterator<X> iterator();
    void add(X x);
}
```

And let's suppose that `Geek` is a subtype of `Person`.

The intuitive expectation is that the following code should work:

```
Collection<Geek> geeks = ... ;
Collection<Person> people = geeks;   //compiler error
for (Person person in people) { ... }
```

This code is, frankly, perfectly reasonable taken at face value. Yet in both Java and Ceylon, this code results in a compiler error at the second line, where the `Collection<Geek>` is assigned to a `Collection<Person>`. Why? Well, because if we let the assignment through, the following code would also compile:

```
Collection<Geek> geeks = ... ;
Collection<Person> people = geeks;   //compiler error
people.add( Person("Fonzie") );
```

We can't let that code by—Fonzie isn't a `Geek`!

Using big words, we say that `Collection` is *nonvariant* in `X`. Or, when we're not trying to impress people with opaque terminology, we say that `Collection` both *produces*—via the `iterator()` method—and *consumes*—via the `add()` method—instances of `X`.

Here's where Java goes off and dives down a rabbit hole, inventing wildcards to try and squeeze a covariant or contravariant type out of a nonvariant type, but mainly succeeding in thoroughly confusing everybody. We're not going to follow Java down the hole.

Instead, we're going to refactor `Collection` into a pure producer interface and a pure consumer interface:

```
interface Producer<out X> {
    Iterator<X> iterator();
}
```

```
interface Consumer<in X> {
    void add(X x);
}
```

Notice that we've annotated the type parameters of these interfaces.

- The `out` annotation specifies that `Producer` is *covariant* in `X`; that it produces instances of `X`, but never consumes instances of `X`.

- The `in` annotation specifies that `Consumer` is *contravariant* in `X`; that it consumes instances of `X`, but never produces instances of `X`.

The Ceylon compiler validates the schema of the type declaration to ensure that the variance annotations are satisfied. If you try to declare an `add()` method on `Producer`, a compilation error results. If you try to declare an `iterate()` method on `Consumer`, you get a similar compilation error.

Now, let's see what that buys us:

- Since `Producer` is covariant in its type parameter `X`, and since `Geek` is a subtype of `Person`, Ceylon lets you assign `Producer<Geek>` to `Producer<Person>`.
- Furthermore, since `Consumer` is contravariant in its type parameter `X`, and since `Geek` is a subtype of `Person`, Ceylon lets you assign `Consumer<Person>` to `Consumer<Geek>`.

We can define our `Collection` interface as a mixin of `Producer` with `Consumer`.

```
interface Collection<X>
        satisfies Producer<X>, Consumer<X> {}
```

Notice that `Collection` remains nonvariant in `X`.

Now, the following code finally compiles:

```
Collection<Geek> geeks = ... ;
Producer<Person> people = geeks;
for (Person person in people) { ... }
```

Which matches our original intuition.

The following code also compiles:

```
Collection<Person> people = ... ;
Consumer<Geek> geekConsumer = people;
geekConsumer.add( Geek("James Gosling") );
```

Which is also intuitively correct—James is most certainly a `Person`!

You're unlikely to spend much time writing your own collection classes, since the Ceylon SDK has a powerful collections framework built in. But you'll still appreciate Ceylon's approach to covariance as a user of the built-in collection types. The collections framework defines two interfaces for each basic kind of collection. For example, there is an interface `List<X>` which represents a read-only view of a list, and is covariant in `X`, and `OpenList<X>`, which represents a mutable list, and is nonvariant in `X`.

1.2.16. Testing the type of an object

Ceylon doesn't have C-style typecasts. Instead, we must test and narrow the type of an object in one step, using the special `if (is ... )` construct. This construct is very, very similar to `if (exists ... )`, which we met earlier.

```
Object object = ... ;
if (is Hello object) {
    object.say();
}
```

The `switch` statement supports a similar construct:

```
Object object = ... ;
switch(object)
case (is Hello) {
    object.say();
}
case (is Person) {
    stream.writeLine(object.firstName);
}
else {
    stream.writeLine($object);
}
```

These constructs protect us from inadvertantly writing code that would cause a `ClassCastException` in Java, just like `if (exists ... )` protects us from writing code that would cause a `NullPointerException`.

Another thing that causes problems when working with generic types in Java is *type erasure*. Generic type arguments are discarded by the compiler, and simply aren't available at runtime. So the following, perfectly sensible, code fragments simply wouldn't compile in Java:

```
if (is List<Person> list) { ... }
```

```
if (is T object) { ... }
```

```
Type<T> tType = T;
```

(Where `T` is a generic type parameter.)

You'll be pleased to know that in Ceylon, these three code fragments compile and function as expected.

1.2.17. Introducing a new type to an object

Ceylon doesn't have multiple inheritance or mixins. If you're used to the single-inheritance-with-interfaces model of Java, you'll find that inheritance in Ceylon works pretty much the same.

However, Ceylon does have something else, that's usually almost as good as, and often better than, true multiple inheritance. An *extension* adds a type, call an *introduced* type, to an existing type, called the *extended* type. An extension doesn't change the original definition of the extended type, and it doesn't affect the internal workings of an object of that type in any way. But from the point of view of a client of the object, the object now has all the attributes and methods of the introduced type, and is assignable to the introduced type.

Let's introduce a type onto every object. Ceylon's `Object` class defines an attribute named `log` of type `Log` that you can use to log stuff. Usually, you use it like this:

```
log.debug("Hello, I'm a debug message");
```

But perhaps you don't like having to type the "`log.`" bit every time you write out a log message, and suppose you don't expect to ever need methods named `info()`, `error()` `warn()` or `debug()` in your application. If so, wouldn't it be nice to just write:

```
debug("Do you think anyone ever reads me?");
```

Obviously, one way to do this would be to copy and paste all the `info()`, `error()` `warn()` and `debug()` methods from `Log` onto `Object`, and have them delegate back to `log`. But we don't like copy/paste programming here.

Another way might be to make `Object` extend `Log`, but that way `Log` would be the logical root of the Ceylon type system, instead of `Object`. That doesn't feel right.

Instead, we're going to introduce `Log` onto `Object` using an extension. If we were able to modify the code of `Object`, this would be as easy as adding an `extension` annotation to the `log`, attribute like this:

```
public extension Log log { ... }
```

But since `Object` is built into the Ceylon SDK, we don't have control over its code, and so we need to take a different approach. We'll write a toplevel *extension method*, sometimes called a *converter*:

```
public extension Log objectToLog(Object o) { return o.log }
```

The extended type is the type of the parameter of the converter. The introduced type is the return type of the converter.

Now, we can't just have hundreds of third-party extensions all vandalizing `Object` with their own introduced types, and polluting the namespace with thousands of introduced methods and attributes. So the client code that uses the extension is responsible for explicitly activating the extension in the compilation unit where it is used. To activate an extension we `import` it.

```
import com.domain.util.objectToLog;
```

```
public class Hello(String? name) {
    ...
    info(greeting);
    ...
}
```

The compiler automatically inserts a call to `objectToLog()`. So the above code is equivalent to:

```
public class Hello(String? name) {
    ...
    objectToLog(this).info(greeting);
    ...
}
```

A compilation unit that doesn't explicitly import `objectToLog` won't be affected by the extension, and won't be able to call the `info()` method without explicitly invoking the `log` attribute.

So you might think that extension are just a trivial trick that saves a few keystrokes of typing effort. But the history of Java demonstrates why extensions are an important feature. Java defines its collections framework in terms of various interfaces, for example, `List`, each with very many methods, mostly convenience methods for the benefit of the user of these interfaces. Because implementing the whole interface from scratch is a daunting task, the collections framework includes abstract classes like `AbstractList` that define a smaller contract for classes that implement the collection interfaces. But there's no way for the collections framework to force you to implement `AbstractList` instead of implementing `List` directly. And, of course, the world is now full of Java code that does implement `List` directly. Which means that introducing a new method to the `List` interface is a huge breaking change that affects a great deal of working, production code.

Ceylon takes a different approach to its collection types. The basic collection interfaces like `List` define a small set of operations. Convenience methods for users of the collections framework are defined by built-in extensions. So we can add new convenience methods whenever we like. And so can you!

Chapter 2. Lexical structure

The lexical structure of Ceylon source files is very similar to Java. Like Java, Unicode escape sequences `\uXXXX` are processed first, to produce a raw stream of Unicode characters. This character stream is then processed by the lexer to produce a stream of terminal tokens of the Ceylon grammar.

2.1. Whitespace

Whitespace characters are the ASCII `SP`, `HT` `FF`, `LF` and `CR` characters.

```
Whitespace := " " | Tab | Formfeed | Newline | Return
```

Outside of a comment, string literal, or single quoted literal, whitespace acts as a token separator and is immediately discarded by the lexer. Whitespace is not used as a statement separator.

2.2. Comments

There are two kinds of comments:

- a multiline comment that begins with `/*` and extends until `*/`, and
- an end-of-line comment begins with `//` and extends until a line terminator: an ASCII `LF`, `CR` or `CR LF`.

Both kinds of comments can be nested.

```
LineComment := "//" ~(Newline|Return)* (Return Newline | Return | Newline)
```

```
MultilineComment := "/*" ( MultilineCommmentCharacter | MultilineComment )* "*/"
```

```
MultilineCommmentCharacter := ~("/"|"*") | ("/" ~"*") => "/" | ("*" ~"/") => "*"
```

The following examples are legal comments:

```
//this comment stops at the end of the line
```

```
/*
   but this is a comment that spans
   multiple lines
*/
```

Comments are treated as whitespace by both the compiler and documentation compiler. Comments may act as token separators, but their content is immediately discarded by the lexer.

2.3. Identifiers and keywords

Identifiers may contain upper and lowercase letters, digits and underscore.

```
IdentifierChar := LowercaseChar | UppercaseChar | Digit
```

```
Digit := "0".."9"
```

```
LowercaseChar := "a".."z" | "_"
```

```
UppercaseChar := "A".."Z"
```

The Ceylon lexer distinguishes identifiers which begin with an initial uppercase character from identifiers which begin with an initial lowercase character or underscore.

```
LIdentifier := LowercaseChar IdentifierChar*
```

```
UIdentifier := UppercaseChar IdentifierChar*
```

The following examples are legal identifiers:

```
Person
```

```
name
```

```
personName
```

```
_id
```

```
x2
```

The following reserved words are not legal identifier names:

```
import class interface alias satisfies extends abstracts in out given void subtype local assign return break throw retry this super if else switch case for fail do while try catch finally exists nonempty is public module package private abstract default fixed override mutable extension deprecated volatile small
```

TODO: Eventually we will probably want to support identifiers in non-European character sets. We can use an initial underscore to distinguish "initial lowercase" identifiers.

2.4. Literals

2.4.1. Numeric literals

A natural number literal has this form:

```
NaturalLiteral = Digit+
```

A floating point number literal has this form:

```
FloatLiteral := Digit+ "." Digit+ ( ("E"|"e") ("+"|"-")? Digit+ )?
```

The following examples are legal numeric literals:

```
69
```

```
6.9
```

```
0.999e-10
```

```
1.0E2
```

The following are *not* valid numeric literals:

```
.33  //Error: floating point literals may not begin with a decimal point
```

```
1.  //Error: floating point literals may not end with a decimal point
```

```
99E+3   //Error: floating point literals with an exponent must contain a decimal point
```

TODO: The symbol `.` is also an operator. Are expressions like `1.decimal` and `1.0.whole` allowed, or do we force the use of parentheses as in `(1).decimal` and `(1.0).whole`?

2.4.2. Character literals

A single character literal consists of a character, surrounded by backticks.

```
CharacterLiteral := "`" Character "`"
```

```
Character := ~("`" | "\" | Tab | Formfeed | Newline | Return | Backspace) | EscapeSequence
```

```
EscapeSequence := "\" ("b" | "t" | "n" | "f" | "r" | "\" | """ | "'" | "`" )
```

The following are legal character literals:

```
`A`
```

```
`#`
```

```
` `
```

```
`\n`
```

TODO: should we support an escape sequence for Unicode character names `\N{name}` like Python does?

2.4.3. String literals

A character string literal is a character sequence surrounded by double quotes.

```
StringLiteral := """ StringCharacter* """
```

```
StringCharacter := ~( "\" | """ | "'" ) | EscapeSequence
```

The following are legal strings:

```
"Hello!"
```

```
" \t\n\f\r,;:"
```

2.4.4. Single quoted literals

Single-quoted strings are used to express literal values for user-defined types. A single quoted literal is a character sequence surrounded by single quotes:

```
QuotedLiteral := "'" StringCharacter* "'"
```

2.5. Operators and delimiters

The following character sequences are operators and/or punctuation:

```
, ; ... { } ( ) [ ] . ?. []. = + - / * % ** ++ -- .. -> ? ! && || => ~ & | ^ == != === < > <= >= <=>
:= .= += -= /= *= %= |= &= ^= ||= &&= ?=
```

Chapter 3. Declarations

All classes, interfaces, methods, attributes and locals must be declared.

3.1. General declaration syntax

All declarations follow a general pattern.

3.1.1. Abstract declaration

Declarations conform to the following general schema:

```
Annotation*
keyword? Type? (TypeName|MemberName) TypeParams? FormalParams*
Supertype?
Interfaces?
TypeConstraints?
Body?
```

The Ceylon compiler enforces identifier naming conventions. Types must be named with an initial uppercase. Members, parameters, locals and packages must be named with an initial lowercase or underscore.

```
PackageName := LIdentifier
```

```
TypeName := UIdentifier
```

```
MemberName := LIdentifier
```

```
ParameterName := LIdentifier
```

Ceylon defines three identifier namespaces:

- classes, interfaces and aliases share a single namespace,
- methods, attributes and locals share a single namespace, and
- packages have their own dedicated namespace.

The Ceylon parser is able to unambiguously identify which namespace an identifier belongs to.

TODO: should we specify that a package name must be an all-lowercase identifier?

3.1.2. Imports and toplevel declarations

A toplevel declaration defines a type—a class, interface or type alias—or a method.

```
ToplevelDeclaration := TypeDeclaration | Method
```

```
TypeDeclaration := Class | Interface | Alias
```

All toplevel declarations with a visibility modifier less strict than `private` must have the same name as the compilation unit filename (after removing the file suffix `.ceylon`). For example, a `public` toplevel class named `Person` must be defined in a file named `Person.ceylon`. A `public` toplevel method named `hello()` must be defined in a file named `hello.ceylon`. Unlike Java, a compilation unit may contain multiple toplevel class or method declarations with the same name.

A compilation unit consists of a list of imported toplevel types and toplevel methods, followed by one or more type or method definitions:

```
Import* ToplevelDeclaration+
```

Each compilation unit belongs to exactly one *package*. An `import` statement allows the compilation unit to refer to a top-level declaration in another package.

```
Import := "import" FullPackageName "." ImportSpec ";"
```

A package is a namespace. A full package name is a period-separated list of initial lowercase identifiers.

```
FullPackageName := PackageName ("." PackageName)*
```

An `import` statement may import either:

- a single (toplevel or member) type,
- a single toplevel method,
- a single named enumerated instance of a class,
- all toplevel declarations of a specified package, or
- all named enumerated instances and member types of a specified class or interface.

```
ImportSpec := TypeSpec | MethodSpec | InstanceSpec | PackageMembersSpec | TypeMembersSpec
```

```
TypeSpec := QualifiedTypeName ("alias" TypeName)?
```

```
MethodSpec := MemberName ("alias" MemberName)?
```

```
InstanceSpec := QualifiedTypeName "." MemberName ("alias" MemberName)?
```

The character `*` acts as a wildcard, just like in Java.

```
PackageMembersSpec := "*"
```

```
TypeMembersSpec := QualifiedTypeName "." "*"
```

The name of a member type must be qualified by the names of its containing types.

```
QualifiedTypeName := (TypeName ".")* TypeName
```

The optional `alias` clause allows resolution of cross-namespace declaration name collisions.

```
import lang.collections.*;
```

```
import transaction.propagation.TxPropagationType.*;
```

```
import java.util.Map.Entry alias MapEntry;
```

```
import my.math.fibonnacciNumber alias fib;
```

```
import my.query.Order.descending alias desc;
```

Code in one compilation unit may refer to a toplevel declaration in another compilation unit in the same package without explicitly importing the declaration. It may refer to a toplevel declaration defined in a compilation unit in another package only if it explicitly imports the declaration.

TODO: Does Ceylon support toplevel attributes? Perhaps just non-`mutable` toplevel attributes?

TODO: Would it be better to specify that there is at most a single `import` statement per source file, for example:

```
import my.query.Order, lang.collections.*, java.util.Map.Entry alias MapEntry;
```

.

TODO: Would it be better to enclose that the imported expressions in single quotes, as they would need to appear in the body of the source file, for example: `'my.query.Order'`, `'lang.collections.*'`, `'java.util.Map.Entry' alias MapEntry`.

3.1.3. Annotation list

Declarations may be preceded by a list of annotations.

```
Annotation := MemberName ( Arguments | Literal+ )?
```

Unlike Java, the name of an annotation may not be a qualified name.

For an annotation with no arguments, or with only literal-valued arguments, the parentheses around, and commas between, the positional arguments may be omitted.

```
doc "The user login action"
by "Gavin King"
   "Andrew Haley"
throws (DatabaseException
        -> "if database access fails")
see (LogoutAction.logout)
scope (session)
action { description="Log In"; url="/login"; }
public deprecated
```

An annotation is an invocation of a toplevel method that occurs when the type is loaded by the virtual machine. The return value of the invocation is made available via reflection.

For example, the built-in `doc` annotation is defined as follows:

```
public
annotation { occurs=onceEachElement; }
Description doc(String description) {
    return Description(description.normalize())
}
```

The annotation may be specified at a program element using any one of three forms.

Using a positional parameter invocation of the method:

```
doc("the name") String name;
```

Using a named parameter invocation of the method:

```
doc {description="the name";} String name;
```

Or using the special abbreviated form for annotations with literal value arguments:

```
doc "the name" String name;
```

And its value may be obtained like this:

```
Description? description = Person.annotations(Description).first;
```

Unlike Java, the same annotation may appear multiple times for the same program element. Furthermore, different annotations (toplevel methods) may return values of the same type.

3.1.4. Type

Method, attribute and formal parameter declarations must declare a type.

A *type* or *type schema* is name (an initial upper case identifier) and an optional list of type parameters, with a set of:

- attribute schemas,

- member method schemas, and
- member class schemas.

Speaking formally:

- An *attribute schema* is a name (an initial lower case identifier) with a type and mutability.
- A *method schema* is a name (an initial lower case identifier) and an optional list of type parameters, with a type (often called the *return type*) and a list of one or more formal parameter lists.
- A *class schema* is a type with a formal parameter list.
- A *formal parameter list* is a list of names (initial lower case identifiers) with types. The *signature* of a formal parameter list is formed by discarding the names, leaving the list of types.

Speaking slightly less formally, we usually refer to an attribute, method, or member class of a type, meaning an attribute schema, member method schema or member class schema.

The *erased signature* of a method or class is formed by:

- taking the signature of the formal parameter list of the member class, or of the first formal parameter list of the method, and
- replacing each occurrence of any type alias in the signature with the aliased type, and
- replacing each occurrence of `Optional<T>` in the signature with `T`, and
- replacing each occurrence of any type parameter in the signature with the first declared upper bound of the type parameter, or `lang.Void` if there is no declared upper bound, and
- replacing the type of any varargs type parameter in the signature with `lang.Void...`, and then
- discarding the type arguments of each element of the list, leaving just the name of a class or interface.

Note: I really, really hate this stuff. Is there any way we can do better than Java here?

Two erased signatures are considered *distinct* if they have different lengths, or if at some position within the lists, the two types are non-identical.

TODO: non-identical, or non-assignable?

TODO: what about varargs parameters? what does Java do?

A type may not have:

- two attributes with the same name,
- a method and an attribute with the same name,
- two methods with the same name and non-distinct erased signatures, or
- two member classes with the same name and non-distinct erased signatures.

A type may be a *subtype* of another type. Subtyping obeys the following rules:

- Identity: `X` is a subtype of `X`.
- Transitivity: if `X` is a subtype of `Y` and `Y` is a subtype of `Z` then `X` is a subtype of `Z`.
- Interfaces are objects: all interfaces are subtypes of `lang.Object`. Conversely, only a subtype of `lang.Object` may satisfy an interface.
- Single root: all types are subtypes of `lang.Void`.

In a certain compilation unit, a type may be *assignable* to another type. If, in some compilation unit, `X` is assignable to `Y`, then:

- For each non-`mutable` attribute of `Y`, `X` has an attribute with the same name, whose type is assignable to the type of the attribute of `Y`.
- For each `mutable` attribute of `Y`, `X` has a `mutable` attribute with the same name and the same type.
- For each method of `Y`, `X` has a method with the same name, with the same number of formal parameter lists, with the same signatures, and whose return type is assignable to the return type of the method of `Y`.
- For each member class of `Y`, `X` has a member class of the same name, with a formal parameter list with the same signature, that is assignable to the member class of `Y`.

Assignability obeys the following rules:

- Identity: `X` is assignable to `X`.
- Subtyping: if `Y` is a subtype of `X` then `Y` is assignable to `X`.
- Transitivity: if `X` is assignable to `Y` and `Y` is assignable to `Z` then `X` is assignable to `Z`.
- Nullsafety: the type `lang.Optional` is not assignable to `lang.Object`. Nor is its subtype `lang.Nothing`.

A type `X` may be assignable to a type `Y` even if `X` is not a subtype of `Y` if there is an enabled extension which introduces `Y` to `X`.

TODO: are we really, truly going to make extensions transitive? That would involve the compiler searching for extension chains. But if not, assignability would not transitive.

Types are identified by the name of the type (a class, interface, alias or type parameter), together with a list of type arguments if the type definition specifies type parameters.

```
Type := ( TypeNameWithArguments ("." TypeNameWithArguments)* | "subtype" ) Abbreviation*
```

If a type has type parameters or a varargs type parameter, a type argument list must be specified. If a type has no type parameters, and no varargs type parameter, no type argument list may be specified.

Unlike Java, the name of a type may not be qualified by the package name.

Certain declarations which usually require an explicit type may omit the type, forcing the compiler to infer it, by specifying the keyword `local` where the type usually appears.

```
InferableType := Type | "local"
```

Type inference is only allowed for block local declarations. The keyword `local` may not be combined with a visibility modifier annotation.

The following type name abbreviations are supported:

- `X?` means `Optional<X>` for any type `X`, and
- `X[]` means `Sequence<X>` for any type `X`.

```
Abbreviation := "?" | "[" "]"
```

Abbreviations may be combined:

```
String?[] words = { "hello", "world", null };
String?? firstWord = words[0];
```

TODO: It might be difficult to support `T??` since `Optional` is erased.

3.1.5. Generic type parameter list

Methods, classes, interfaces and aliases may declare one or more generic type parameters.

```
TypeParams := "<" TypeParam ("," TypeParam)* VarargsTypeParam ">"
```

A declaration with type parameters is called *generic* or *parameterized*.

- A class or interface declaration with no type parameters defines exactly one type. A parameterized class or interface declaration defines a template for producing types: one type for each possible combination of type arguments that satisfy the type constraints specified by the class or interface. The types of members of the this type are determined by replacing every appearance of each type parameter in the schema of the parameterized type definition with its type argument.
- A method declaration with no type parameters defines exactly one operation per type. A parameterized method declaration defines a template for producing overloaded operations: one operation for each possible combination of type arguments that satisfy the type constraints specified by the method declaration.
- A class declaration with no type parameters defines exactly one instantiation operation. A parameterized class declaration defines a template for producing overloaded instantiation operations: one instantiation operation for each possible combination of type arguments that satisfy the type constraints specified by the class declaration. The type of the object produced by an instantiation operation is determined by substituting the same combination of type arguments for the type parameters of the parameterized class.

A type parameter is itself a type, visible within the body of the declaration it parameterizes. A type parameter is a subtype of every upper bound of the type parameter. However, a class, interface, or alias may not extend or implement a type parameter.

Each type parameter has a name and a specified *variance*.

```
TypeParam := Variance TypeName
```

A *covariant* type parameter is indicated using `out`. A *contravariant* type parameter is indicated using `in`.

```
Variance :=  ("out" | "in")?
```

A type parameter declared neither `out` nor `in` is called *nonvariant*.

```
Map<K, V>
```

```
Sender<in M>
```

```
Container<out T>
```

```
BinaryFunction<in X, in Y, out R>
```

TODO: Would `produces` and `consumes` be better?

For a generic type `T<X>`, a type `A`, and a subtype `B` of `A`:

- If `X` is a covariant type parameter, `T<B>` is a subtype of `T<A>`.
- If `X` is a contravariant type parameter, `T<A>` is a subtype of `T<B>`.
- If `X` is nonvariant (neither covariant nor contravariant), there is no subtype relationship between `T<A>` and `T<B>`.

A covariant type parameter may only appear in covariant positions of the type definition. A contravariant type parameter may only appear in contravariant positions of the type definition. Nonvariant type parameters may appear in any position.

- The return type of a method is a covariant position.
- A formal parameter type of a method is a contravariant position.

- A type parameter of a method is a contravariant position.
- A formal parameter type of a member class initializer is a contravariant position.
- A type parameter of a member class is a contravariant position.
- The type of a non-`mutable` attribute is a covariant position.
- The type of a `mutable` attribute is a nonvariant position.
- An upper bound of a type parameter in a contravariant position is a contravariant position.
- An upper bound of a type parameter in a covariant position is a covariant position.
- A lower bound of a type parameter in a contravariant position is a covariant position.
- A type parameter in a covariant position cannot have a lower bound.
  TODO: is this correct?
- A covariant type argument of a satisfied or extended type is a covariant position.
- A contravariant type argument of a satisfied or extended type is a contravariant position.
- A covariant type argument of a type in a covariant position is a covariant position.
- A contravariant type argument of a type in a covariant position is a contravariant position.
- A covariant type argument of a type in a contravariant position is a contravariant position.
- A contravariant type argument of a type in a contravariant position is a covariant position.
- A nonvariant type argument of a type is a nonvariant position.
- A formal parameter of a callable parameter in a contravariant position is covariant.
- A formal parameter of a callable parameter in a covariant position is contravariant.
- The return type of a callable parameter in a contravariant position is contravariant.
- The return type of a callable parameter in a covariant position is covariant.

These rules apply to members declared by the type, and to members inherited from supertypes.

Every Ceylon class or interface has an implicit type parameter that never needs to be declared. This special type parameter, referred to using the keyword `subtype`, represents the concrete type of the current instance (the instance that is being invoked). It is considered a covariant type parameter of the type, and may only appear in covariant positions of the type definition. It is upper bounded by the type (and is therefore assignable to the type).

```
public interface Wrapper<out X> {}
```

```
public abstract class Wrappable() {
   public Wrapper<subtype> wrap() {
       return Wrapper(this)
   }
}
```

```
public class Special() extends Wrappable() {}
```

```
Special special = Special();
Wrapper<Special> wrapper = special.wrap();
```

The only expression assignable to the type `subtype` is the special value `this`, except inside the body of a method or attribute annotated `fixed`, where the class that declares the method or attribute is assignable to `subtype`.

TODO: should we let you declare type constraints on `subtype`?

A *varargs type parameter*, identified by an elipsis ... accepts a list of zero or more type arguments.

```
VarargsTypeParam := TypeName "..."
```

Varargs type parameters are always non-variant.

Inside the declaration of the parameterized type or method, a varargs type parameter may be used as a type argument to other types which accept a varargs type parameter, or it may be used as the type of the last formal parameter declared by a method. It may not appear in any other position. The varargs type parameter acts as a pseudo-type. It is treated by the Ceylon compiler as if it were a type with no members, to which no other type may be assigned, and which can only by assigned to itself.

```
Method<X, T, P...>
```

TODO: should we let you declare a variant varargs type parameter? This is useful for modelling tuples.

TODO: should we let you write things like:

```
class Foo<P...>(Tuple<Bar<P>...> barTup) {
    ...
    void baz(Baz<P>... bazs) {
        ...
    }
}
```

It's probably only really useful if we also let you do stuff like this:

```
List<T> zip<T,P...>(List<P>... x, T producing(P... args)) {
    return from (Natural i in 0..min(x.lastIndex...)) select (f(x[i]...))
}
```

3.1.6. Type argument list

A list of *type arguments* produces a new type from a parameterized type, or a new method schema from a method schema with type parameters.

```
TypeNameWithArguments := TypeName TypeArguments?
```

A type argument list is a list of types, and an optional varargs type argument.

```
TypeArguments := "<" Type ("," Type)* ("," VarargsType)? ">"
```

```
VarargsType := TypeName "..."
```

A type argument list *conforms* to a type parameter list if:

- a type argument that satisfies the constraints of the type parameter is specified for every type parameter, and.
- if the type parameter list has no varargs type parameter, then there are no additional type arguments, and no varargs type argument.

```
Entry<String,Person>
```

```
Stack<Frame>.Entry
```

```
Callable<X,T,P..>
```

A type argument is substituted for every appearance of the corresponding type parameter in the schema of the parameterized type definition, including:

- attribute types,
- method return types,

- method formal parameter types,
- initializer formal parameter types, and
- type arguments of extended classes and satisfied interfaces.

```
Map<Key, Value>
```

In the case of a varargs type parameter:

- the type arguments are appended to the list of type arguments in every parameterized type in which the varargs type parameter appears, and
- a list of formal parameters whose types are the type arguments is appended to the list of formal parameters of every method declaration in which the varargs type parameter appears.

```
Method<Order, Item, Product prod, Natural quantity>
```

TODO: should we let you fill in the formal parameter names, so you can call this thing using named parameters?

```
Method<Order, Item, Product prod, Natural quantity>
```

A type argument may itself be a parameterized type or type parameter.

```
Map<Key, List<Item>>
```

Substitution of type arguments may result in an ambiguity:

- two methods of the parameterized type with the same name may now also have non-distinct erased signatures, or
- two member classes of the parameterized type with the same name may now also have non-distinct erased signatures.

In this case, the member class or method may not be called. Any invocation of the member results in a compiler error.

A type with an ambiguity may never be extended or implemented by another type. It may not appear in an `extends`, `satisfies` or `abstracts` clause.

Type arguments are *reified* in Ceylon. An instance of a generic type holds a reference to its type arguments. Therefore, the following are legal in Ceylon:

- testing the runtime value of a type argument of an instance, for example, `objectList is List<Person>` or `case (is List<Person>)`,
- filtering exceptions based on type arguments, for example, `catch (NotFoundException<Person> pnfe)`,
- testing the runtime value of an instance against a type parameter, for example `x is T`, or against a type with a type parameter as an argument, for example, `objectList is List<T>`,
- obtaining a `Type` object representing a type with type arguments, for example, `List<Person>`,
- obtaining a `Type` object representing the runtime value of a type parameter, for example, `T`, or of a type with a type parameter as an argument, for example, `List<T>`,
- obtaining a `Type` object representing the runtime value of a type argument of an instance using reflection, for example, `objectList.type.arguments.first`, and
- instantiating a type parameter with an initialization parameter specification, for example, `T(parent)`.

Varargs type parameters are not reified. None of the above operations can be performed with a vararg type parameter.

3.1.7. Formal parameter list

Method and class declarations may declare formal parameters, including defaulted parameters and a varargs parameter.

```
FormalParams :=
"("
FormalParam ("," FormalParam)* ("," DefaultParam)* ("," VarargsParam)? |
DefaultParam ("," DefaultParam)* ("," VarargsParam)? |
VarargsParam?
")"
```

```
FormalParam := Param | EntryParamPair | RangeParamPair
```

Each parameter is declared with a type and name and may have annotations and/or parameters of its own.

```
Param := Annotation* (Type|"void") ParameterName FormalParams*
```

A parameter with its own parameter list (or lists) is called a *callable parameter*. Think of it as an abstract local method that must be defined by the caller when the method is invoked or the class is instantiated.

```
(String label, void onClick())
```

```
(Comparison by(X x, X y))
```

A callable parameter declaration is equivalent to a formal parameter declaration with no parameter lists where the type is the callable type of the method declaration. So the above are equivalent to:

```
(String label, Callable<Object> onClick)
```

```
(Callable<Comparison,X,X> by)
```

Defaulted parameters specify a default argument.

```
DefaultParam := FormalParam Specifier
```

The `=` specifier is used throughout the language to indicate a value which cannot be reassigned.

```
Specifier := "=" Expression
```

Defaulted parameters must occur after non-defaulted parameters in the formal parameter list.

```
(Product product, Natural quantity=1)
```

The type of the default argument expression must be assignable to the declared type of the formal parameter.

The elipsis `...` indicates that a formal parameter is either:

- A *varargs parameter*, which accepts a list of arguments of the specified type `T`, or a single argument of type `T[]`. Inside the method, the varargs parameter has type `T[]`.
- A *tuple parameter* representing a list of parameters whose types are defined by a varargs type parameter. Inside the method, the argument has the pseudo-type of the varargs type parameter and is assignable to any tuple parameter of the same pseudo-type.

```
VarargsParam := Annotation* Type "..." ParameterName
```

The varargs parameter or tuple parameter must be the last formal parameter in the list.

```
(Name name, Organization? org=null, Address... addresses)
```

```
(T instance, P... args)
```

Parameters of type `Entry` or `Range` may be specified as a pair of variables.

```
EntryParamPair := Annotation* Type ParameterName "->" Type ParameterName
```

```
RangeParamPair := Annotation* Type ParameterName ".." ParameterName
```

A variable pair declaration of form `U u -> V v` results in a single parameter of type `Entry<U,V>`.

```
(Key key -> Value value)
```

A variable pair declaration of form `T x .. y` results in a single parameter of type `Range<T>`.

```
(Float value, Integer min..max)
```

A formal parameter may not be be declared `mutable`, and may not be assigned to within the body of the method or class.

3.1.8. Extended class

A class may extend another class using the `extends` clause.

```
Supertype := "extends" Type PositionalArguments
```

A class may extend only one superclass. If the superclass is a parameterized type, the `extends` clause must specify type arguments.

```
extends Person(name, org)
```

Suppose `X` and `Y` are classes.

- If `X` extends `Y`, then `X` is a subtype of `Y`.
- If `X` extends `Y<B>`, then `X` is a subtype of `Y<B>`.
- If `X<T>` extends `Y<T>`, then `X<B>` is a subtype of `Y<B>` for any type `B`.
- If `X<T>` extends `Y`, then `X<B>` is a subtype of `Y` for any type `B`.

A user-defined class must extend `lang.Object` or one of its subclasses.

3.1.9. Satisfied interfaces

Classes and interfaces may satisfy (implement or extend) interfaces, using the `satisfies` clause.

```
Interfaces = "satisfies" Type ("," Type)*
```

A class or interface may satisfy multiple interfaces. If a satisfied interface is a parameterized type, the `satisfies` clause must specify type arguments.

```
satisfies T[], Collection<T>
```

Suppose `Y` is an interface and `X` is a class or interface.

- If `X` satisfies `Y`, then `X` is a subtype of `Y`.
- If `X` satisfies `Y<B>`, then `X` is a subtype of `Y<B>`.
- If `X<T>` satisfies `Y<T>`, then `X<B>` is a subtype of `Y<B>` for any type `B`.
- If `X<T>` satisfies `Y`, then `X<B>` is a subtype of `Y` for any type `B`.

3.1.10. Generic type constraint list

Method, class and interface declarations which declare generic type parameters may declare constraints upon the type parameters using the `given` clause.

```
TypeConstraints = ("given" TypeConstraint)+
```

```
TypeConstraint := TypeName FormalParams? Interfaces? Subtypes?
```

```
Subtypes := "abstracts" Type ("," Type)*
```

There are three kinds of type constraint:

- an upper bound, `given X satisfies T`, specifies that the type parameter `X` is a subtype of a given type `T`,
- a lower bound, `given X abstracts T`, specifies that a given type `T` is a subtype of the type parameter `X`,
- an initialization parameter specification, `X(...)` specifies that the type parameter `X` is a class with the given formal parameter types.

A constraint may not refer to a varargs type parameter.

Initialization parameter specifications allow instantiation of the generic type.

A constraints affects the type arguments that can be assigned to a type parameter:

- A type argument to a type parameter with an upper bound must be a type which is a subtype of all upper bounds.
- A type argument to a type parameter with a lower bound must be a type of which all lower bounds are subtypes.

A constraint affects the assignability of a type parameter:

- A type parameter is considered a subtype of its upper bound.
- The lower bound of a type parameter is considered a subtype of the type parameter.

```
given X satisfies Number<X>
given Y(Natural count) satisfies Number<Y>
```

```
given T satisfies Ordinal, Comparable<T> abstracts X
```

TODO: Should the type parameter upper bound default to `Object` instead of `Void`?

TODO: Should we move `in` and `out` down to the `given` clause?

TODO: should we let you declare a constraint for a varargs type parameter?

```
class Foo<P...>(P... args)
        given P satisfies Comparable<P> {
    ...
}
```

3.2. Interfaces

An *interface* is a type schema. Interfaces do not specify the implementation of their members and may not be directly instantiated.

```
Interface :=
Annotation*
"interface" TypeName TypeParams?
Interfaces?
TypeConstraints?
InterfaceBody
```

```
InterfaceBody := "{" AbstractDeclaration* "}"
```

The body of an interface contains:

- member (method, attribute and member class) declarations, and

- nested interface declarations.

```
AbstractDeclaration := AbstractMethod | AbstractAttribute | TypeDeclaration
```

Interface method and attribute declarations may not specify implementation.

```
public interface Comparable<T> {
    Comparison compare(T other);
}
```

Interface members inherit the visibility modifier of the interface.

TODO: Refine this. Consider block-local interface declarations.

TODO: if methods of interfaces can define defaulted parameters, precisely how do we implement that?

3.2.1. Interface inheritance

An interface may extend any number of other interfaces.

```
public interface List<T>
        satisfies T[], Collection<T> {
    ...
}
```

The types listed after the `satisfies` keyword are the supertypes. All supertypes of an interface must be interfaces. An interface may not extend the same interface twice (not even with distinct type arguments).

The semantics of interface inheritance are exactly the same as Java. An interface inherits all members (methods, attributes and member types) of every supertype.

The schema of the inherited members is formed by substituting type arguments specified in the `satisfies` clause.

3.3. Classes

A *class* is a stateful, instantiable type. It is a type schema, together with implementation details of the members of the type.

```
Class :=
Annotation*
"class" TypeName TypeParams? FormalParams
Supertype?
Interfaces?
TypeConstraints?
ClassBody
```

```
ClassBody := "{" Instances? (Declaration | Statement)* "}"
```

The body of a class contains:

- member (method, attribute and member class) declarations,
- nested interface declarations,
- instance initialization code, and,
- optionally, a list of enumerated named instances of the class.

```
Declaration := Method | SimpleAttribute | AttributeGetter | AttributeSetter | TypeDeclaration
```

A class may be annotated `mutable`. If a class is not annotated `mutable` it is called an *immutable type*, and it may not:

- declare or inherit `mutable` attributes,
- extend a `mutable` superclass, or

- implement an interface annotated `mutable`.

Ordinarily, a declaration that occurs in a block of code is a block local declaration—it is visible only to statements and declarations that occur later in the same block. This rule is relaxed for certain declarations that occur directly inside a class body:

- declarations with explicit visibility modifiers—whose visibility is determined by the modifier, and
- declarations that occur in the second part of the body of the class, after the last statement of the initializer—which are visible to all other declarations in the second part of the body of the class.

3.3.1. Class initializer

Ceylon classes do not support a Java-like constructor declaration syntax. Instead:

- The body of the class declares *initialization parameters*. An initialization parameter may be used anywhere in the class body, including in method and attribute definitions.
- The initial part of the body of the class is called the *initializer* and contains a mix of declarations, statements and control structures. The initializer is executed every time the class is instantiated.

An initialization parameter may be used to specify or initialize the value of an attrbute:

```
public class Key(Lock lock) {
    public Lock lock = lock;
}
```

```
public class Counter(Natural start=0) {
    public mutable Natural count := start;
    public void inc() { count++; }
}
```

An initialization parameter may even be used within the body of a method, attribute getter, or attribute setter:

```
public class Key(Lock lock) {
    public Lock lock { return lock }
}
```

```
public class Key(Lock lock) {
    public void lock() { lock.engage(this); }
    public void unlock() { lock.disengage(this); }
}
```

A subclass must pass values to each superclass initialization parameter in the `extends` clause.

```
public class SpecialKey1()
        extends Key( SpecialLock() ) {
    ...
}
```

```
public class SpecialKey2(Lock lock)
        extends Key(lock) {
    ...
}
```

The class initializer is responsible for initializing the state of a new instance of the class, before a reference to the new instance is available to clients.

```
public abstract class Point() {
    public Decimal x;
    public Decimal y;
}
```

```
public class DiagonalPoint(Decimal distance)
        extends Point() {

    Decimal pos = sqrt(distance**2/2) * distance.sign;
    x = pos;
```

```
    y = pos;

    assert ("must have distance " distance " from origin")
        that ( x**2 + y**2 == distance**2 );

}
```

An initializer may invoke, evaluate or assign members of the current instance of the class (the instance being initialized) without explicitly specifying the receiver.

An initializer of a member class may invoke, evaluate or assign members of the current instance of the containing class (the instance upon which the constructor was invoked) without explicitly specifying the receiver.

A class may be declared inside the body of a method or attribute, in which case the initializer may refer to any non-`mutable` local, block local attribute getter or block local method declared earlier within the containing scope. It may not refer to `mutable` locals from the containing scope.

The following restrictions apply to statements and declarations that appear within the initializer of the class:

- They may not evaluate attributes or invoke methods that are declared later in the body of the class upon the current object or `this`.
- They may not pass `this` as an argument of a method invocation or the value of an attribute assignment.
- They may not declare an abstract method or attribute.
- They may not declare a `default` method or attribute.

The remainder of the body of the class consists purely of declarations, including abstract and `default` methods and attributes. It may not directly contain statements or control structures, but may freely use `this`, and may invoke any method or evaluate any attribute of the class. The usual restriction that a declaration may only be used by code that appears later in the block containing the declaration is relaxed. The declarations in this section may not contain specifiers or initializers (`=` or `:=`).

Superclass members may be invoked, evaluated or assigned anywhere inside the body of the class. The superclass initializer is executed before the subclass initializer.

TODO: should class initialization parameters be allowed to be declared `public/package/module`, allowing a shortcut simple attribute declaration like in Scala?

The *callable type* of a class captures the type and formal parameter types of the class. The callable type is `Callable<T,P...>`, where `T` is the class and `P...` are the formal parameter types of the class. A varargs parameter is considered of type `T[]` where `T` is the declared vararg type.

3.3.2. Class inheritance

A class may extend another class, and implement any number of interfaces.

```
public mutable
class Customer(Name name, Organization? org = null)
        extends Person(name, org) {
    ...
}
```

```
class Token()
        extends Datetime()
        satisfies Comparable<Token>, Identifier {
    ...
}
```

The types listed after the `satisfies` keyword are the implemented interfaces. The type specified after the `extends` keyword is a superclass. A class may not implement the same interface twice (not even with distinct type arguments).

If a class does not explicitly specify a superclass using `extends`, its superclass is `lang.Object`. There are one exception to this rule: the built-in class `lang.Void` which does not have a superclass.

The semantics of class inheritance are exactly the same as Java. A class:

- inherits all members (methods, attributes, and member types) of its superclass, except for members that it *overrides*,
- must declare or inherit a member that overrides each member of every interface it implements directly or indirectly, unless the class is declared `abstract`, and
- must declare or inherit a member that overrides each abstract member of its superclass, unless the class is declared `abstract`.

The schema of the inherited members is formed by substituting type arguments specified in the `extends` clause.

Furthermore, the initializer of the superclass is always executed before the initializer of the subclass whenever the subclass is instantiated.

3.3.3. Class instance enumeration

The keyword `case` is used to specify an enumerated named instance of a class. All `case`s must appear in a list as the first line of a class definition.

```
Instances := Instance ("," Instance)* ("..." | ";")
```

```
Instance := Annotation* "case" MemberName Arguments?
```

If the `case` list ends in `;`, the instance list is called *closed*. If the `case` list ends in `...`, the instance list is called *open*.

If a class has a closed instance list, the class may not:

- be instantiated
- have subclasses, or
- have members annotated `default`.

A class may not specify enumerated named instances if it:

- is annotated `abstract`,
- has generic type parameters, or
- is nested directly or indirectly inside another class or inside a block.

```
public class DayOfWeek() {
    case sun,
    case mon,
    case tues,
    case wed,
    case thurs,
    case fri,
    case sat;
}
```

TODO: Should we make the parens on the class declaration optional in this case: a closed instance list with no parameters?

```
public class DayOfWeek(String name) {

    doc "Sunday"
    case sun("Sunday"),

    doc "Monday"
    case mon("Monday"),

    doc "Tuesday"
    case tues("Tuesday"),

    doc "Wednesday"
    case wed("Wednesday"),
```

```
    doc "Thursday"
    case thurs("Thursday"),

    doc "Friday"
    case fri("Friday"),

    doc "Saturday"
    case sat("Saturday");

    public String name = name;

}
```

```
public class TransactionPropagation(void inProgress(), void notInProgress()) {

    case required {
        void inProgress() {}
        void notInProgress() {
            tx.begin();
        }
    },

    case supports {
        void inProgress() {}
        void notInProgress() {}
    },

    case mandatory {
        void inProgress() {}
        void notInProgress() {
            throw TransactionMandatory()
        }
    },

    case notSupported {
        void inProgress() {
            throw TransactionNotSupported()
        }
        void notInProgress() {}
    },

    case requiresNew {
        void inProgress() {
            throw TransactionRequiresNew()
        }
        void notInProgress() {
            tx.begin();
        }
    };

    public void propagate(Transaction tx) {
        if (tx.inProgress) {
            inProgress();
        }
        else {
            notInProgress();
        }
    }

}
```

A class with declared `cases` implicitly implements `lang.Selector`.

Enumerated instances of a class are instantiated when the class is loaded by the virtual machine, with the specified arguments.

TODO: should we have the ability to declare a restricted list of member subclasses using `case class`, for example:

```
abstract class Node(String name) {

    case root("root"),
    case class Branch(String name, Node parent) extends Node(name) { ... },
    case class Leaf(String name, Node parent) extends Node(name) { ... };

    ...

}
```

```
Node root = Node.root;
Node branch = Node.Branch("Furry", root);
```

```
Node leaf = Node.Leaf("Kittens", branch);
```

3.3.4. Overloaded classes

Multiple toplevel classes belonging to the same package, or multiple member classes of the same containing class may declare the same name. The classes are called *overloaded*. Overloaded classes:

- must extend and overload a common *root* type,
- must have distinct erased signatures,
- may not declare default parameters, and
- except for the root type, may not declare any member with a visibility modifier.

Then the class name always refers to the root type, except in instantiations. In the case of instantiation, the correct overloaded class is resolved at compile time, using the mechanism that Java uses to choose between overloaded constructors.

3.3.5. Overriding member classes

A member class annotated `abstract` or `default` may be overridden by subclasses of the class which contains the member class. The subclass must declare a member class:

- annotated `override`,
- with the same name as the member class it overrides,
- that extends the member class it overrides, and
- with a formal parameter list with the same signature as the member class it overrides, after substitution of type arguments specified in the `extends` or `satisfies` clause.

Finally, the overridden member class must be visible to the member class annotated `override`.

Then instantiation of the member class is polymorphic, and the actual subtype instantiated depends upon the concrete type of the containing class instance.

By default, the member class annotated `override` has the same visibility modifier as the member class it overrides. The member class may not declare a stricter visibility modifier than the member class it overrides.

3.4. Methods

A *method* is a callable block of code. Methods may have parameters and may return a value.

```
Method := MethodHeader ( Block | Specifier? ";" )
```

All method declarations specify the method name and one or more formal parameter lists. A method declaration may specify a type, called the *return type*, to which the values the method returns is assignable, or it may specify that the method is a `void` method—a method which does not return a value. The return type of a `void` method is considered to be `Void`.

```
MethodHeader := Annotation* (InferableType | "void") MemberName TypeParams? FormalParams+ TypeConstraints?
```

A method implementation may be specified using either:

- a block of code, or
- a reference to another method.

A member method body may invoke, evaluate or assign members of the current instance of the class which defines the method (the instance upon which the method was invoked) without explicitly specifying the receiver.

A member method body of a member class may invoke, evaluate or assign members of the current instance of the containing class (the containing instance of the instance upon which the method was invoked) without explicitly specifying the receiver.

A toplevel method body may not refer to `this` or `super`, since there is no current instance.

A method may be declared inside the body of another method or attribute, in which case it may refer to any non-`mutable` local, block local attribute getter or block local method declared earlier within the containing scope. It may not refer to `mutable` locals from the containing scope.

The Ceylon compiler preserves the names of method parameters.

3.4.1. Method implementation

A method body may be a block. If the method is a `void` method, the block may not contain a `return` directive that specifies an expression. Otherwise, every conditional execution path of the block must end in a `return` directive that specifies an expression assignable to the return type of the method.

```
public Integer add(Integer x, Integer y) {
    return x + y
}
```

```
Identifier createToken() {
    return Token()
}
```

```
public void print(Object... objects) {
    for (Object object in objects) {
        log.info($object);
    }
}
```

```
public void addEntry(V key -> U value) {
    map.define(key,value);
}
```

```
public Set<T> singleton<T>(T element)
        given T satisfies Comparable<T> {
    return TreeSet(element)
}
```

Note that a method which declares the return type `Void` is not a `void` method. A method with declared type `Void` must return a value of type `Void` (any value will do).

```
void say(String) { ... }

void hello() {
    say("hello");
}

Void goodbye() {
    return say("goodbye")
}
```

A block local method with a single `return` directive may be declared using the keyword `local` in place of the explicit return type declaration. The type of the method is inferred to be the type of the returned expression.

```
local add(Integer x, Integer y) {
    return x + y
}
```

3.4.2. Callable type of a method

The *callable type* of a method captures the return types and formal parameter types of the method.

- The callable type of a method with a single parameter list is `Callable<T,P...>` where `T` is the declared type of the method, or `Void` if the method is `void`, and `P...` are the formal parameter types of the method.

- The callable type of a method with multiple parameter lists is `Callable<O,P...>`, where `O` is the callable type of a method produced by eliminating the first formal parameter list, and `P...` are the formal parameter types of the first formal parameter list of the method.

A varargs parameter is considered of type `T[]` where `T` is the declared vararg type.

A method body may be an expression that evaluates to a callable object, specified using `=`. The type of the callable object must be assignable to the callable type of the method.

```
Float say(String words) = person.say;
```

```
Comparison order(String x, String y) = getOrder();
```

A method may declare multiple lists of formal parameters. A method which declares more than one formal parameter list returns instances of `Callable`, usually method references.

```
Comparison getOrder()(Natural x, Natural y) {
    Comparison order(Natural x, Natural y) { return x<=>y }
    return order
}
```

A method body may *only* refer to parameters in the first parameter list. It may not refer to parameters of other parameter lists. Parameters declared by parameter lists other than the first parameter list are not considered visible inside the body of the method.

The type of a callable object returned by a method with multiple parameter lists must be assignable to the callable type of a method formed by taking the method with multiple parameter lists and eliminating its first parameter list.

A block local method which specifies a callable object expression may be declared using the keyword `local` in place of the explicit return type declaration. The return type of the method is inferred to be the type of the type argument to the second type parameter of the expression type `Callable` (the return type).

```
local sqrt(Integer x) = 2.root(x);
```

3.4.3. Overloaded methods

A class may declare or inherit multiple methods with the same name. The methods are called *overloaded*. Overloaded methods:

- must have distinct erased signatures, and
- may not declare default parameters.

A class may not not declare of inherit a method with the same name as an attribute it declares or inherits.

Like Java, Ceylon resolves overloaded methods at compile time.

3.4.4. Interface methods and abstract methods

If there is no method body in a method declaration, the implementation of the method must be specified later in the block, or the class that declares the method must be annotated `abstract`. If no implementation is specified, the method is considered an *abstract method*.

```
public U? get(V? key);
```

Methods declared by interfaces never specify an implementation:

```
AbstractMethod := MethodHeader ";"
```

3.4.5. Overriding methods

Method overriding is the foundation of polymorphism in Ceylon:

- A method annotated `default` may be overridden by subclasses of the class or interface which declares the method.
- A method annotated `fixed` *must* be overridden by every subclass of the class or interface which declares the method.
- An interface method or abstract method must be overridden by every non-`abstract` class that is assignable to the interface or abstract class type, unless the class inherits a non-abstract method from a superclass which overrides the interface method or abstract method.

To override a method, a class must declare a method:

- annotated `override`,
- with the same name as the method it overrides,
- the same number of formal parameter lists, with the same signatures, as the method it overrides, after substitution of type arguments specified in the `extends` or `satisfies` clause, and
- with a return type that is assignable to the return type of the method it overrides, after substitution of type arguments specified in the `extends` or `satisfies` clause.

Finally, the overridden method must be visible to the method annotated `override`.

Then invocation of the method is polymorphic, and the actual method invoked depends upon the concrete type of the class instance.

By default, the method annotated `override` has the same visibility modifier as the method it overrides. The method may not declare a stricter visibility modifier than the method it overrides.

```
abstract public class AbstractSquareRooter() {
    public Float squareRoot(Float x);
}
```

```
class ConcreteSquareRooter()
        extends AbstractSquareRooter() {
    override Float squareRoot(Float x) { ... }
}
```

For abstract methods, a special shortcut form of overriding is permitted. A subclass initializer may simply specify an instance of `Callable` as the implementation of the abstract method declared by the superclass. No formal parameter list, return type declaration, or `override` annotation is necessary.

```
class ConcreteSquareRooter()
        extends AbstractSquareRooter() {
    Float sqrt(Float x) { ... }
    squareRoot = sqrt;
}
```

Toplevel methods cannot be overridden, and so toplevel method invocation is never polymorphic.

TODO: are you allowed to override the default value of a defaulted parameter?

TODO: are you required to have the same formal parameter names in the two methods? I don't see that this would be necessary. In a named parameter invocation, you just use the names declared by the member of the compile-time type, and they are mapped positionally to the parameters of the overriding method.

3.5. Attributes

There are three kinds of declarations related to *attribute* definition:

- Simple attribute declarations define state (very similar to a Java field or local variable).
- Attribute getter declarations define how the value of a derived attribute is obtained.
- Attribute setter declarations define how the value of a derived attribute is assigned.

Unlike Java fields, Ceylon attribute access is polymorphic and attribute definitions may be overridden by subclasses.

All attributes have a type and name. The type of the attribute is specified by the simple attribute declaration or attribute getter declaration. An attribute may be `mutable`, in which case its value can be assigned using the `:=` and compound assignment operators. This is the case for simple attributes explicitly annotated `mutable`, or for attributes with a setter declaration.

```
AttributeHeader := Annotation* InferableType MemberName
```

An attribute body may invoke, evaluate or assign members of the current instance of the class which defines the method (the instance upon which the attribute was invoked) without explicitly specifying the receiver.

An attribute body of a member class may invoke, evaluate or assign members of the current instance of the containing class (the containing instance of the instance upon which the attribute was invoked) without explicitly specifying the receiver.

An attribute may be declared inside the body of another method or attribute, in which case it may refer to any non-`mutable` local, block local attribute getter or block local method declared earlier withing the containing scope. It may not refer to `mutable` locals from the containing scope.

3.5.1. Simple attributes and locals

A simple attribute defines state.

```
SimpleAttribute := AttributeHeader (Specifier | Initializer)? ";"
```

A simple attribute or local annotated `mutable` represents a value that can be assigned multiple times. A simple attribute or local not annotated `mutable` represents a value that can be specified exactly once.

The value of a non-`mutable` attribute is specified using `=`. A `mutable` attribute may be initialized using the assignment operator `:=`.

```
Initializer := ":=" Expression
```

Formal parameters of classes and methods are also considered to be simple attributes.

```
mutable Natural count := 0;
```

```
public Integer max = 99;
```

```
public Decimal pi = calculatePi();
```

A simple attribute declared directly inside the body of a class represents state associated with the instance of the class. Repeated evaluation of the attribute of a particular instance of the class returns the same result until the attribute of the instance is assigned a new value.

A *local* represents state associated with execution of a particular block of code. A local is really just a special case of a simple attribute declaration, but one whose state is not held across multiple executions of the block of code in which the local is defined.

- A simple attribute declared inside a block (the body of a method, attribute getter or attribute setter) is a local.
- A block local simple attribute declared inside the body of a class is a local if it is not used inside a method, attribute setter or attribute getter declaration.
- A formal parameter of a class is a local if it is not used inside a method, attribute setter or attribute getter declaration.
- A formal parameter of a method is a local.

A local is a block local declaration—it is visible only to statements and declarations that occur later in the same block or class body, and therefore it may not declare a visibility modifier.

The semantics of locals are identical to Java local variables.

The compiler is permitted to optimize block local simple attributes to a simple Java field declaration or local variable. Block local attributes may not be accessed via reflection.

A block local simple attribute with a specifier or initializer may be declared using the keyword `local` in place of the explicit type declaration. The type of the local or attribute is inferred to be the type of the specifier or initializer expression.

```
local names = List<String>();
```

```
mutable local count:=0;
```

3.5.2. Attribute getters

An attribute getter is a callable block of code with no parameters, that returns a value.

```
AttributeGetter := AttributeHeader Block
```

An attribute getter defines how the value of a derived attribute is obtained.

```
public Float total {
    Float sum := 0.0;
    for (LineItem li in lineItems) {
        sum += li.amount;
    }
    return sum
}
```

If an attribute getter has a matching attribute setter, we say that the attribute is `mutable`. Otherwise we say it is non-`mutable`.

A block local attribute getter with a single `return` directive may be declared using the keyword `local` in place of the explicit type declaration. The type of the local or attribute is inferred to be the type of the returned expression.

```
local name {
    return Name(firstName, initial, lastName)
}
```

3.5.3. Attribute setters

An attribute setter is a callable block of code that accepts a single value and does not return a value.

```
AttributeSetter := Annotation* "assign" MemberName Block
```

An attribute setter defines how the value of a derived attribute is assigned. Every attribute setter must have a corresponding getter with the same name.

```
public String name { return join(firstName, lastName) }
public assign name { firstName = first(name); lastName = last(name); }
```

TODO: should we require that the corresponding getter be annotated `mutable`?

TODO: should we allow overloaded attribute setters, for example:

```
assign Name name { firstName = name.firstName; lastName = name.lastName; }
```

3.5.4. Interface attributes and abstract attributes

If there is no specifier, initializer or getter implementation, the value or implementation of the attribute must be specified later in the block, or the class that declares the attribute must be annotated `abstract`. If no value or implementation is specified, the attribute is considered an *abstract attribute*.

```
package mutable String firstName;
```

Attributes declared by interfaces never specify an initalizer, getter or setter:

```
AbstractAttribute := AttributeHeader ";"
```

3.5.5. Overriding attributes

Ceylon allows attributes to be overridden, just like methods:

- An attribute annotated `default` may be overridden by subclasses of the class or interface which declares the method.
- A method annotated `fixed` *must* be overridden by every subclass of the class or interface which declares the method.
- An interface attribute or abstract attribute must be overridden by every non-`abstract` class that is assignable to the interface or abstract class type, unless the class inherits a non-abstract attribute from a superclass which overrides the interface attribute or abstract attribute.

A non-`mutable` attribute may be overridden by a simple attribute or attribute getter. A `mutable` attribute may be overridden by a `mutable` simple attribute or by an attribute getter and setter pair.

A class which overrides an attribute must declare an attribute:

- annotated `override`,
- with the same name as the attribute it overrides,
- with a type that is assignable to the type of the attribute it overrides, after substitution of type arguments specified in the `extends` or `satisfies` clause,
- or with *exactly the same type* as the attribute it overrides, after substitution of type arguments specified in the `extends` or `satisfies` clause, if the attribute it overrides is `mutable`, and
- that is `mutable`, if the attribute it overrides is `mutable`.

Finally, the overridden attribute must be visible to the attribute annotated `override`.

A non-`mutable` attribute may be overridden by a `mutable` attribute.

TODO: is that really allowed? It could break the superclass. Should we say that you are allowed to do it when you implement an interface attribute, but not when you override a superclass attribute?

Then evaluation and assignment of the attribute is polymorphic, and the actual attribute evaluated or assigned depends upon the concrete type of the class instance.

By default, the attribute annotated `override` has the same visibility modifier as the attribute it overrides. The method may not declare a stricter visibility modifier than the method it overrides.

```
abstract module class AbstractPi() {
    module Float pi;
}
```

```
class ConcretePi()
        extends AbstractPi() {
    override Float pi { ... }
}
```

For abstract attributes, a special shortcut form of overriding is permitted. A subclass initializer may simply specify or assign a value to the attribute declared by the superclass. No type declaration or `override` annotation is necessary.

```
class ConcretePi()
        extends AbstractPi() {
    Float calculatePi() { ... }
    pi = calculatePi();
}
```

3.6. Type aliases

A *type alias* allows a type to be referred to more compactly.

```
Alias := Annotation* "alias" TypeName TypeParams? Interfaces? TypeConstraints? ";"
```

A type alias may satisfy either a single interface or a single class.

The alias type is assignable to the satisfied type, and the satisfied type is assignable to the alias type.

```
public alias People satisfies List<Person>;
```

A shortcut is provided for definition of private aliases.

```
import java.util.List alias JavaList;
```

Type aliases are not reified types. The metamodel reference for a type alias—for example, `People`— returns the metamodel object for the aliased type—in this case, `List<Person>`.

TODO: could we reify them? This would let us define type aliases that satisfy multiple interfaces. For example:

```
package alias ComparableCollection<X> satisfies Collection<X>, Comparable<X>;
```

3.7. Declaration modifiers

In Ceylon, all declaration modifiers are annotations.

3.7.1. Summary of compiler instructions

The following annotations are compiler instructions:

- `public`, `module`, `package` and `private` determine the visibility of a declaration (by default, the declaration is visible only to statements and declarations that appear later inside the same block).
- `abstract` specifies that a class cannot be instantiated.
- `default` specifies that a method, attribute, or member class may be overridden by subclasses.
- `override` indicates that a method, attribute, or member type overrides a method, attribute, or member type defined by a supertype.
- `mutable` specifies that an attribute or local may be assigned, or that a class has assignable attributes.
- `fixed` specifies that a method or attribute must be overridden by every subclass.
- `extension` specifies that a method or attribute getter is a converter, or that a class is a decorator.
- `deprecated` indicates that a method, attribute or type is deprecated. It accepts an optional `String` argument.
- `volatile` indicates a volatile simple attribute.

TODO: should it be called `overrides`?

The following annotation is a hint to the compiler that lets the compiler optimize compiled bytecode for non-64 bit architectures:

- `small` specifies that a value of type `Natural`, `Integer` or `Float` contains 32-bit values.

By default, `Natural`, `Integer` and `Float` are assumed to represent 64-bit values.

The annotation names in this section are treated as keywords by the Ceylon compiler. This is a performance optimization to minimize the need for lookahead in the parser.

TODO: Should we require an `abstract` modifier for abstract methods and attributes of abstract classes like Java does?

3.7.2. Visibility and name resolution

Classes, interfaces, aliases, methods, attributes, locals, formal parameters and type parameters have names. Occurrence of a name in code implies a hard dependency from the code in which the name occurs to the schema of the named declaration. We say that a class, interface, alias, method, attribute, formal parameter or type parameter is *visible* to a certain program element if its name may occur in the code that defines that program element.

- A formal parameter or type parameter is never visible outside the declaration it belongs to.
- Any declaration that occurs inside a block (the body of a method, attribute getter or attribute setter) is not visible to code outside the block.

The visibility of any other declaration depends upon its *visibility modifier*, if any. By default:

- a declaration that occurs directly inside a class body is not visible to code outside the class definition, and
- a toplevel declaration is not visible to code outside the package containing its compilation unit.

The visibility of a declaration with a visibility modifier annotation is determined by the visibility modifier:

- `private` specifies that the declaration is visible to all code in the same compilation unit,
- `package` specifies that the declaration is visible to all code in any compilation unit in the same package,
- `module` specifies that the declaration is visible to all code in any package in the same module,
- `public` specifies that the declaration is visible to all code in any module.

The visibility of a nested declaration may not be less strict than the visibility of the body (class body, interface body, or block) which contains it. For example, a `package`-visibility class may not contain a `public`-visibility attribute, method, or member class. Likewise, a block may not declare a visibility modifier, so a declaration that occurs inside a block may not declare a visibility modifier. Note, however, that this restriction does not apply to the visibility of nested declarations annotated `override` which inherit their visibility from the declaration they override.

The class `lang.Visibility` defines the visibility levels:

```
public class Visibility {

    doc "A program element visible to all
         compilation units."
    case public,

    doc "A program element visible to
         compilation units in the same
         module."
    case module,

    doc "A program element visible to
         compilation units in the same
         package."
    case package,

    doc "A program element visible to
         the compilation unit in which
         its is declared."
    case private,

    doc "A program element local to the
         block in which it is defined."
    case block;

}
```

The following declarations define the visibility modifier annotations:

```
doc "The |public| visibility modifier
     annotation."
annotation { occurs=onceEachElement; }
```

```
public Visibility public() {
    return public
}
```

```
doc "The |module| visibility modifier
     annotation."
annotation { occurs=onceEachElement; }
public Visibility module() {
    return module
}
```

```
doc "The |package| visibility modifier
     annotation."
annotation { occurs=onceEachElement; }
public Visibility package() {
    return package
}
```

```
doc "The |private| visibility modifier
     annotation."
annotation { occurs=onceEachElement; }
public Visibility private() {
    return private
}
```

TODO: how are we going to go about compiling these classes which define the reserved-word annotations? A special compiler switch to turn off these reserved words? (Seems reasonable.)

3.7.3. Extensions

An extension allows values of one type to be transparently converted to values of another type. Extensions are declared by annotating a method, attribute or class `extension`. An extension must be:

- a toplevel method with exactly one formal parameter,
- a member method with no formal parameters,
- a toplevel class with exactly one initialization parameter,
- a member class with no initialization parameters, or
- an attribute.

An toplevel extension method is called a *converter*. An toplevel extension class is called a *decorator*.

A decorator may not use the special value `this` in expressions. This allows the compiler to optimize away the actual instance of the class in certain circumstances.

TODO: do we really need this restriction in order to enable a mere optimization?

Extensions apply to a certain *extended type*:

- for toplevel extension methods, the extended type is the declared type of the formal parameter,
- for member extension methods, the extended type is the type that declares the extension method,
- for toplevel extension classes, the extended type is the declared type of the initialization parameter,
- for member extension classes, the extended type is the type that contains the member class, and
- for extension attributes, the extended type is the type that declares the attribute.

Extensions define an *introduced type*:

- for extension methods, the introduced type is the declared return type of the method,
- for extension classes, the introduced type is the class, and

- for extension attributes, the introduced type is the declared type of the attribute.

The introduced type may not be a `mutable` type.

```
public extension Log objectToLog(Object object) {
    return object.log;
}
```

```
public class Person(User user) {
    ...
    public extension User user = user;
    ...
}
```

```
public extension class SequenceUtils<N>(N[] seq)
        given N extends Number, Comparable<N> {

    public N[] positiveElements() {
        return seq.elements() where (N n) (n>0)
    }

    public N[] elementsLessThan(N limit) {
        return seq.elements() where (N n) (n<limit)
    }

    ...
}
```

Note: I actually much prefer the readability of `User personToUser(extends Person person)` and `SequenceUtils<T>(extends T[] collection)`, but this doesn't work for attributes and member methods.

We say that an extension class or toplevel method is *enabled* in a compilation unit if the class or toplevel method is imported by that compilation unit.

```
import org.domain.app.extensions.objectToLog;
import org.domain.utils.SequenceUtils;
```

A wildcard `.*`-style import may not be used to import an extension.

An extension attribute, member class or member method is enabled in every compilation unit.

If an extension is enabled in a compilation unit, the extended type is assignable to the introduced type in that compilation unit.

```
import org.mydomain.myproject.extensions.objectToLog;
...
Person person = ...;
User user = person;
info("person is a User and this is a Log!");
```

```
import org.domain.utils.SequenceUtils;
...
Integer[] zeroToOneHundred = 0..100;
Integer[] oneToNine = zeroToOneHundred.positiveElements().elementsLessThan(10);
```

An introduced type may result in an ambiguity:

- the introduced type may have a member type with the same name as a member type of the extended type, or of some other introduced type, and the two member types may have non-distinct erased signatures,
- the introduced type may have a method with the same name as a method of the extended type, or of some other introduced type, and the two methods may have non-distinct erased signatures,
- the introduced type may have an attribute with the same name as an attribute or method of the extended type, or of some other introduced type,
- the introduced type may have a method with the same name as an attribute of the extended type, or of some other introduced type.

In this case, the member type, method or attribute may not be called. Any invocation or evaluation of the member results in a compiler error.

TODO: We should allow an introduced type to specify `override (ExtendedType.member)` or `override (OtherIntroducedType.member)` to resolve an ambiguity like this.

TODO: We need to make the following idiom work:

```
public extension Foo toFoo(Bar bar) {
    if (is Foo bar) {
        return bar //it is already a Foo, use its customized implementation
    }
    else {
        return FooImpl(bar) //provide an implementation of Foo
    }
}
```

When an invocation of an introduced method or evaluation of an introduced attribute is executed, or when an instance of the extended type is assigned to a program element of the introduced type, the extension is invoked to produce an instance of the introduced type that will receive the invocation or evaluation.

TODO: extensions are nice, and quite powerful, but they aren't enough to implement an embedded query language like in JPA. Dynamic languages let you implement a method to respond to an unknown member invoked at runtime. Java 6 lets you do a similar thing at compile time using a processor (a compiler plugin). I think we can have the best of both worlds and let you write an extension method that returns a set of members to be introduced to the extended type.

3.7.4. Annotation constraints

The following meta-annotations provide information to the compiler about the annotations upon which they appear. They are applied to a toplevel method declaration that defines an annotation.

- `inherited` specifies that the annotation is automatically inherited by subtypes.
- `annotation` specifies constraints upon the occurence of an annotation. By default, an annotation may appear multiple times on any program element.

The meta-annotation `annotation` accepts the following parameters.

- `occurs` specifies that the annotation may occur at most once in a certain scope. Its accepts one argument of type `Occurrence`: `onceEachElement`, `onceEachType`.
- `of` specifies the kinds of program element at which the annotation occurs. Its accepts one or more arguments of type `Element`: `classes`, `interfaces`, `methods`, `attributes`, `aliases`, `parameters`.
- `withType` specifies that the annotation may only be applied to types that are assignable to the specified type, to attributes or parameters of the specified type, or to methods with the specified return type.
- `withParameterTypes` specifies that the annotation may only be applied to methods with the specified formal parameter types.
- `withAnnotation` specifies that the annotation may only be applied to program elements at which the specified annotation occurs.

```
public
annotation {
    of = classes;
    occurs = onceEachType;
}
Entity entity(LockMode lockMode) {
    return Entity(lockMode)
}
```

```
public
annotation {
    of = { attributes, parameters };
    withType = String;
    occurs = onceEachElement;
}
```

```
PatternValidator pattern(Regex regex) {
    return PatternValidator(regex)
}
```

TODO: Should `annotation` be required for toplevel methods which can be used as annotations?

3.7.5. Documentation compiler

The following annotations are instructions to the documentation compiler:

- `doc` specifies the description of a program element.
- `by` specifies the authors of a program element.
- `see` specifies a related member or type.
- `throws` specifies a thrown exception type.

The `String` arguments to the `deprecated`, `doc`, `throws` and `by` annotations are parsed by the documentation compiler as Seam Text, a simple ANTLR-based wiki text format.

These annotations are defined by the package `doc`:

```
inherited annotation { occurs = onceEachElement; }
public Description doc(String description) {
    return Description(description.normalize())
}
```

```
annotation { occurs = onceEachElement; }
public Author[] by(String... authors) {
    return from (String author in authors)
                select Author(author.normalize())
}
```

```
public RelatedElement see(ProgramElement pe) {
    return Related(pe)
}
```

```
public RelatedElement see(ProgramElement pe -> String description) {
    return Related(pe, description.normalize())
}
```

```
public Related throws(Type<Exception> type) {
    return ThrownException(type)
}
```

```
public ThrownException throws(Type<Exception> type -> String description) {
    return ThrownException(type, description.normalize())
}
```

Chapter 4. Blocks and control structures

Method, attribute and class bodies contain procedural code that is executed when the method or attribute is invoked, or when the class is instantiated. The code contains expressions and control directives and is organized using blocks and control structures.

4.1. Name resolution

An unqualified identifier (an identifier not preceded by `.`, `[].` or `?.`) that appears in a program element refers to a declaration elsewhere: a class, interface, alias, method, attribute, local or enumerated instance.

A *body* is a block, class body or interface body. A declaration is *in scope* at a program element if:

- it is a formal parameter, type parameter, or control structure variable of a body that contains the program element, or
- it occurs earlier in a body that contains the program element, or
- it occurs in the second part of a class body, after the last statement or declaration of the initializer, and the program element is contained in the second part of the class body, or
- it is a declaration imported by the compilation unit containing the program element and is visible to the program element, or
- it is a toplevel declaration in the package containing the program element and is visible to the program element.

If there is no declaration with the specified name in scope at the program element where the specified name occurs, a compilation error occurs.

If multiple declarations with the specified name are in scope at the program element where the specified name occurs, the name refers to the declaration which is not *hidden* by another declaration:

- if an inner body is contained (directly or indirectly) an outer body, a declaration, formal parameter, type parameter or control structure variable of the inner body hides a declaration, formal parameter, type parameter or control structure variable in the outer body,
- a formal parameter of a class body hides an attribute of the class,
- a declaration occurring in a body containing the program element hides a declaration imported by the compilation unit, and
- a declaration imported by the compilation unit hides a toplevel declaration of the package containing the program element.

If there are multiple unhidden declarations with the specified name, and they are not all overloaded declarations of the same method or class, the name is ambiguous, and a compilation error occurs.

A body may not contain two declarations with the same name unless they are overloaded versions of the same method or class, or unless one is a formal parameter of the class body and the other is an attribute of the class. A package may not contain two toplevel declarations with the same name unless they are overloaded versions of the same method or class.

Note that this code is not legal:

```
String uppercase(String string) {
    String string = string.uppercase;  //compiler error!
    return string;
}
```

However, this code *is* legal, since class initialization parameters and attributes may share a name:

```
public class Person(String name) {
    public String name = name;
}
```

Note that this code is not legal:

```
Entry<Float,Float> xy() {
    Float x { return y }  //compiler error!
    Float y { return x }
    return x->y
}
```

Nor is this code legal, since all three statements occur inside the initializer of the class:

```
class Point() {
    Float x { return y }  //compiler error!
    Float y { return x }
    Entry<Float,Float> xy = x->y;
}
```

However, this code *is* legal, since the statements do not occur in the initializer of the class:

```
public class Point() {
    Float x { return y }
    Float y { return x }
}
```

4.2. Blocks and statements

A *block* is list of semicolon-delimited statements, method, attribute and local declarations, and control structures, surrounded by braces. Some blocks end in a control directive. The statements, local specifiers and control structures belonging to a block are executed sequentially in the order that they appear inside the block. Execution of a block begins at the first statement, local specifier, or control structure of the block. Execution of a block terminates when the last statement, local specifier, or control structure of the block finishes executing, when a control directive that terminates the block is executed, or when an exception is thrown.

```
Block := "{" (Declaration | Statement)* DirectiveStatement? "}"
```

A *statement* is an assignment or specification, an invocation of a method, an instantiation of a class, a control structure or a control directive.

```
Statement := ExpressionStatement | Specification | ControlStructure
```

A simple attribute or local may not be used in an expression until its value has been explicitly specified or initialized. The Ceylon compiler guarantees this by evaluating all conditional branches that lead to the first use of an attribute of local in an expression. Each conditional branch must specify or assign a value to the simple attribute or local before using it in an expression.

Every simple attribute of a non-`abstract` class must be explicitly specified or initialized by the initializer of the class or by the initializer of one of its superclasses. The Ceylon compiler guarantees this by evaluating all conditional branches that lead to termination of the initializer without an uncaught exception. Each conditional branch must specify or assign a value to the simple attribute before the initializer terminates without an uncaught exception.

A simple attribute or local may not be the target of a specifier expression if its value has already been specified. The Ceylon compiler guarantees this by evaluating all conditional branches that lead to the use of a simple attribute or local in a specifier expression. No conditional branch may specify a value to the simple attribute or local before using it in a specifier expression.

TODO: chapter 16 of the JLS goes into way more detail about the notion of "definite assignment" and "definite unassignment". We eventually need something more like this level of detail, I suppose.

TODO: should we allow blocks and control structures to be annotated, for example:

```
doc "unsafe assignment"
suppressWarnings(typesafety)
do {
    apple = orange;
}
```

```
doc "synchronize on the lock"
```

```
try (lock) { ... }
```

4.2.1. Expression statements

Only certain expressions are valid statements: assignment, prefix or postfix increment or decrement, invocation of a method and instantiation of a class.

```
ExpressionStatement := ( Assignment | IncrementOrDecrement | Invocation ) ";"
```

For example:

```
x := 1;
```

```
x++;
```

```
log.info("Hello");
```

```
Main(p);
```

TODO: it would be possible to say that any expression is a valid statement, but this seems to just open up more potential programming errors. So I think it's better to limit statements to assignments, invocations and instantiations.

TODO: should we let you leave off the ; on the last expression statement in the block, like we do for directives?

4.2.2. Control directives

Control directive statements end execution of the current block and force the flow of execution to resume in some outer scope. They may only appear at the end of a block, so the semicolon terminator is optional.

```
DirectiveStatement := Directive ";"?
```

```
Directive := Return | Throw | Break | Continue | Retry
```

Ceylon provides the following control directives:

- the `return` directive—to return a value from a method or attribute getter,
- the `break` directive—to terminate a loop,
- the `continue` directive—to jump to the next iteration of a loop,
- the `throw` directive—to raise an exception, and
- the `retry` directive—to re-execute a `try` block, reinitializing all resources.

```
throw Exception()
```

```
retry
```

```
return x+y
```

```
return "Hello"
```

```
break true
```

```
continue
```

The `return` directive may not appear outside the body of a method or attribute getter. In the case of a `void` method, no expression may be specified. In the case of a non-`void` method or attribute getter, an expression must be specified. The expression type must be assignable to the return type of the method. When the directive is executed, the expression is evalu-

ated to determine the return value of the method or attribute getter.

```
Return := "return" Expression?
```

The `break` directive may not be appear outside the body of a loop. If the loop has no `fail` block, the `break` directive may not specify an expression. If the loop has a `fail` block, the `break` directive must specify an expression of type `lang.Boolean`. When the directive is executed, the expression is evaluated and a value of `false` specifies that the `fail` block should be executed.

```
Break := "break" Expression?
```

The `continue` directive may not be appear outside the body of a loop.

```
Continue := "continue"
```

A `throw` directive may appear anywhere and may specify an expression of type `lang.Exception`. When the directive is executed, the expression is evaluated and the resulting exception is thrown. If no expression is specified, the directive is equivalent to `throw Exception()`.

```
Throw := "throw" Expression?
```

A `retry` directive may not appear outside of a `catch` block.

```
Retry := "retry"
```

4.2.3. Specification statements

A specification statement may specify the value of a non-`mutable` attribute or local that has already been declared earlier in the block. It may even specify a value of type `Callable` for a method that was declared earlier in the block.

```
Specification := MemberName Specifier ";"
```

The Ceylon language distinguishes between assignment to a `mutable` value (the `:=` operator) and specification of the value of a non-`mutable` local or attribute (using `=`). A specification is not an expression.

The specified expression type must be assignable to the type of the attribute or local.

A specification may appear inside an control structure, in which case the compiler validates that all paths result in a properly specified method or attribute. For example:

```
String description;
Comparison order(X x, X y);
if (reverseOrder()) {
    description = "Reverse order";
    order = Order.reverse;
}
else {
    description = "Natural order";
    order = Order.natural;
}
```

4.2.4. Nested declarations

Blocks may contain declarations. A declaration that occurs in a block is a block local declaration—it is visible only to statements and declarations that occur later in the same block, and therefore it may not declare a visibility modifier.

TODO: Note that Java does not let you define an interface inside a method, so we should either add the same restriction, or figure out workaround. Note that Java doesn't let you define a method inside a method either, but we can wrap the nested method in an anonymous class. Note that Java has a funny semantic for a class nested inside an interface nexte

TODO: Note that Java has a funny semantic for a class nested inside an interface nested inside another class. All nested classes of interfaces are considered static inner classes, so this class does not have access to the members of the containing class (unintuitively). We need to figure out how to treat this case. Also consider the related case of a class nested inside an interface nested inside a method.

4.3. Control structures

Control of execution flow may be achieved using control directives and control structures. Control structures include conditionals, loops, and exception management.

Ceylon provides the following control structures:

- the `if/else` conditional—for controlling execution based upon a boolean value, and dealing with null values,
- the `switch/case/else` conditional—for controlling execution using an enumerated list of values,
- the `while` and `do/while` loops—for loops which terminate based upon the value of a boolean expression,
- the `for/fail` loop—for looping over elements of a collection, and
- the `try/catch/finally` exception manager—for managing exceptions and controlling the lifecycle of objects which require explicit destruction.

```
ControlStructure := IfElse | SwitchCaseElse | While | DoWhile | ForFail | TryCatchFinally
```

Control structures are not considered to be expressions, and therefore do not evaluate to a value.

TODO: Should we support, like Java, single-statement control structure bodies without the braces.

4.3.1. Control structure variables

Some control structures allow embedded declaration of a *variable* that is available inside the control structure body.

```
Variable := Type MemberName
```

In certain cases, the explicit type be omitted, forcing the compiler to infer it, by specifying the keyword `local` where the type usually appears. The type of the variable is inferred to be the type of the expression that follows.

```
InferableVariable := InferableType MemberName
```

A variable is treated as a formal parameter of the control structure body.

4.3.2. Control structure conditions

Some control structures expect conditions:

- a *boolean condition* is satisfied when a boolean expression evaluates to `true`,
- an *existence condition* is satisfied when an expression of type `Optional<X>` evaluates to a non-null value,
- a *nonemptiness condition* is satisfied when an expression of type `Optional<Container>` evaluates to a non-null, non-empty value, and
- an *assignabilty condition* is satisfied when an expression evaluates to an instance of a type assignable to the specified type.

TODO: does `is` really test assignability, or should it it test that the value is an instance of a subtype of the given type?

```
Condition := Expression | ExistsCondition | IsCondition
```

```
ExistsCondition := ("exists" | "nonempty") (InferableVariable Specifier | Expression)
```

```
IsCondition := "is" (Variable Specifier | Type Expression)
```

Any condition contains an expression. In the case of existence, nonemptiness and assignability conditions, the expression may be a specifier of a variable declaration.

The type of a condition expression depend upon whether the `exists`, `nonempty` or `is` modifier appears:

- If no `is`, `exists` or `nonempty` modifier appears, the condition must be an expression of type `Boolean`.
- If the `is` modifier appears, the condition expression may be of any type.
- If the `exists` modifier appears, the condition expression must be of type `Optional<X>` for some type `X`.
- If the `nonempty` modifier appears, the condition must be an expression of type `Optional<Container>`.

For `exists` or `nonempty` conditions:

- If the condition declares a variable, the variable must be declared of type `X`, where the specifier expression is of type `Optional<X>`.
- If the condition expression is a local, the local will be treated by the compiler as having type `X` inside the block that follows immediately, where the conditional expression is of type `Optional<X>`.

Note: I suppose we could have a `nonempty` condition on a `Sequence<X>` let you treat `seq.first` and `seq.last` as type `X` instead of `X?`, but the danger is that the list could change underneath us. And we do already have the idiom `if (exists X first = seq.first) { doSomething(first); }` which is only a little more verbose than `if (nonempty seq) { doSomething(seq.first); }`

For `is` conditions:

- If the condition declares a variable, the specifier expression type need not be assignable to the declared type of the variable, or
- if the condition is a local, the local will be treated by the compiler as having the specified type inside the block that follows immediately.

The semantics of a condition depend upon whether the `exists`, `nonempty` or `is` modifier appears:

- If no `is`, `exists` or `nonempty` modifier appears, the condition is satisfied if the expression evaluates to `true` when the control structure is executed.
- If the `is` modifier appears, the condition is satisfied if the expression evaluates to an instance of a class that is assignable to the specified type when the control structure is executed.
- If the `exists` modifier appears, the condition is satisfied if the expression evaluates to an instance of `Something<X>` when the control structure is executed.
- If the `nonempty` modifier appears, the condition is satisfied if the expression evaluates to an instance of `Something<Container>` for which `value.empty` evaluates to `false` when the control structure is executed.

Note that these are formal definitions. In fact, the compiler erases `Optional<T>` to `T` before generating bytecode. So `if (exists x)` is actually processed as `if (x!=null)` by the virtual machine.

TODO: should we have a new kind of condition which directly compares types, allowing you to treat multiple variables as having the tested type inside the block, for example:

```
List<X> xs = ...;
X x = ...;
if (X satisfies Comparable<X>) {
    X x0;
    if (exists X first = xs.first) {
        x0 = max(x, first);
    }
    else {
        x0 = x;
    }
    return forAll(X y in xs) every (x<=x0)
}
```

```
String string<P...>(Tuple<P...> tuple) {
    if (T, Q... satisfies P...) {
        return $tuple.head + ", " + string(tuple.tail);
```

```
    }
    else {
        return ";"
    }
}
```

4.3.3. `if/else`

The `if/else` conditional has the following form:

```
IfElse := If Else?
```

```
If := "if" "(" Condition ")" Block
```

```
Else := "else" (Block | IfElse)
```

When the construct is executed, the condition is evaluated. If the condition is satisfied, the `if` block is executed. Otherwise, the `else` block, if any, is executed.

For example:

```
if (payment.amount <= account.balance) {
    account.balance -= payment.amount;
    payment.paid := true;
}
else {
    throw NotEnoughMoneyException()
}
```

```
public void welcome(User? user) {
    if (exists user) {
        log.info("Hi " user.name "!");
    }
    else {
        log.info("Hello World!");
    }
}
```

```
public Payment payment(Order order) {
    if (exists Payment p = order.payment) {
        return p
    }
    else {
        return Payment(order)
    }
}
```

```
if (exists Payment p = order.payment) {
    if (p.paid) {
        log.info("already paid");
    }
}
```

```
if (is CardPayment p = order.payment) {
    return p.card
}
```

4.3.4. `switch/case/else`

The `switch/case/else` conditional has the following form:

```
SwitchCaseElse := Switch ( Cases | "{" Cases "}" )
```

```
Switch := "switch" "(" Expression ")"
```

```
Cases := CaseItem+ DefaultCaseItem?
```

```
CaseItem := "case" "(" Case ")" Block
```

```
DefaultCaseItem := "else" Block
```

The `switch` expression may be of any type. The `case` values must be expressions of type assignable to `Case<X>`, where `X` is the `switch` expression type. Alternatively, the `case`s may be assignability conditions.

```
Case := Expression ("," Expression)* | "is" Type
```

If no `else` block is specified, the `switch` expression type must be a class with a closed enumerated instance list, and all enumerated `case`s of the class must be explicitly listed.

If the `switch` expression type is a class with a closed enumerated instance list, and all enumerated `case`s of the class are explicitly listed, no `else` block may be specified.

When the construct is executed, the `switch` expression is evaluated, and the resulting value is tested against the `case` values using `Case.test()`. The `case` block for the first `case` value that tests `true` is executed. If no `case` value tests `true`, and an `else` block is defined, the `else` block is executed.

For an assignability condition `case`, if the `switch` expression is a local, then the local will be treated by the compiler as having the specified type inside the `case` block.

For example:

```
public PaymentProcessor processor {
    switch (payment.type)
    case (null) {
        throw NoPaymentTypeException()
    }
    case (credit, debit) {
        return cardPaymentProcessor
    }
    case (check) {
        return checkPaymentProcessor
    }
    else {
        return interactivePaymentProcessor
    }
}
```

```
switch (payment.type) {
    case (credit, debit) {
        log.debug("card payment");
        cardPaymentProcessor.process(payment, user.card);
    }
    case (check) {
        log.debug("check payment");
        checkPaymentProcessor.process(payment);
    }
    else {
        log.debug("other payment type");
    }
}
```

```
switch (payment) {
    case (is CardPayment) {
        pay(payment.amount, payment.card);
    }
    case (is CheckPayment) {
        pay(payment.amount, payment.check);
    }
    else {
        log.debug("other payment type");
    }
}
```

TODO: should it be a `catch`-style syntax instead of `case (is ...)`?

4.3.5. `for/fail`

The `for/fail` loop has the following form:

```
ForFail := For Fail?
```

```
For := "for" "(" ForIterator ")" Block
```

```
Fail := "fail" Block
```

An iteration variable declaration must specify one or two iteration variables, and an iterated expression that contains the range of values to be iterated.

```
ForIterator := InferableVariable ("->" InferableVariable)? "in" Expression
```

The type of the iterated expression depends upon the iteration variable declarations:

- The iterated expression must be an expression of type assignable to `Iterable<X>` where `X` is the declared type of the iteration variable.
- If two iteration variables are defined, the iterated expression type must be assignable to `Iterable<Entry<U,V>>` where `U` and `V` are the declared types of the iteration variables.

When the construct is executed:

- the iterator is obtained by calling `iterator()`, and then
- the `for` block is executed once for each value of type `X` produced by the iterator, until the iterator is exhausted.

Note that:

- if the iterated expression is also of type `Sequence<X>`, the compiler is permitted to optimize away the use of `Iterator`, instead using indexed element access.
- if the iterated expression is a range constructor expression, the compiler is permitted to optimize away creation of the `Range`, and generate the indices using the `successor` operation.

If the loop exits early via execution of one of the control directives `break true`, `return` or `throw`, the `fail` block is not executed. Otherwise, if the loop completes, or if the loop exits early via execution of the control directive `break false`, the `fail` block, if any, is executed.

For example:

```
for (Person p in people) {
    log.info(p.name);
}
```

```
mutable Float sum := 0.0;
for (Integer i in -10..10) {
    sum += x[i];
}
```

```
for (Natural month -> Float temp in monthlyTempsList) {
    plot(month,temp);
}
```

```
for (String word -> Natural freq in wordFrequencyMap) {
    log.info("The frequency of the " word " is " freq ".");
}
```

```
for (Person p in people) {
    log.debug("Testing person: " p.name ".");
    if (p.age >= 18) {
        log.info("Found an adult: " p.name ".");
        break true
    }
}
fail {
    log.info("no adults");
}
```

4.3.6. `while` and `do/while`

The `while` loop has the form:

```
While := LoopCondition Block
```

The `do/while` loop has the form:

```
DoWhile := "do" Block LoopCondition ";"
```

The loop condition determines when the loop terminates.

```
LoopCondition := "while" "(" Condition ")"
```

When the construct is executed, the block is executed repeatedly, until the loop condition first evaluates to `false`, at which point iteration ends. In a `while` loop, the block is executed after the condition is evaluated. In a `do/while` loop, the second block is executed before it is evaluated.

TODO: does `do/while` need a fail block? Python has it, but what is the real usecase?

For example:

```
mutable Natural n:=0;
OpenList<Natural> seq = ArrayList<Natural>();
while (n<=max) {
    seq.append(n);
    n+=step(n);
}
```

```
Iterator<Person> iter = org.employees.iterator();
while (exists Person p = iter.head) {
    log.info(p.name);
    iter.=tail;
}
```

```
mutable Person person := ....;
do {
    log.info(person.name);
    person.=parent;
}
while (!person.dead);
```

4.3.7. `try/catch/finally`

The `try/catch/finally` exception manager has the form:

```
TryCatchFinally := Try Catch* Finally?
```

```
Try := "try" ("(" Resource ")")? Block
```

```
Catch := "catch" "(" Variable ")" Block
```

```
Finally := "finally" Block
```

The type of each `catch` variable declaration must be assignable to `lang.Exception`.

The `try` block may declare a *resource* expression. A resource expresson produces a heavyweight object that must be released when execution of the `try` terminates. Each resource expression must be of type assignable to `lang.Usable`.

```
Resource := InferableVariable Specifier | Expression
```

When the construct is executed:

- the resource expression, if any, is evaluated, and then `begin()` is called upon the resulting resource instance, then

- the `try` block is executed, then
- `end()` is called the resource instance, if any, with the exception that propagated out of the `try` block, if any, then
- if an exception did propagate out of the `try` block, the first `catch` block with a variable to which the exception is assignable, if any, is executed, and then
- the `finally` block, if any, is executed.

For example:

```
try ( File file = File(name) ) {
    file.open(readOnly);
    ...
}
catch (FileNotFoundException fnfe) {
    log.info("file not found: " name "");
}
catch (FileReadException fre) {
    log.info("could not read from file: " name "");
}
finally {
    if (file.open) file.close();
}
```

```
try (semaphore) {
    if ( !map.defines(key) ) {
        map[key] := value;
    }
}
```

(This example shows a Ceylon-ified version of the Java `synchronized` block.)

```
try ( Transaction() ) {
    try ( Session s = Session() ) {
        return s.get(Person, id)
    }
    catch (NotFoundException e) {
        return null
    }
}
```

The `retry` directive re-evaluates the resource expression, if any, and then re-executes the `try` block, calling `begin()` and `end()` upon the resource instance.

```
mutable Natural retries := 0;
try ( Transaction() ) {
    ...
}
catch (TransactonTimeoutException tte) {
    if (retries < 3) {
        retries++;
        retry
    }
    else {
        throw tte
    }
}
```

Chapter 5. Expressions

Ceylon expressions are significantly more powerful than Java, allowing a more declarative style of programming.

Expressions are formed from:

- literal values, special values, and metamodel references,
- enumerated instance references,
- callable references,
- invocation of methods and instantiation of classes,
- evaluation and assignment of attributes,
- enumeration of sequences, and
- operators.

An *atom* is a literal or special value, an enumerated sequence of expressions, or a parenthesized expression.

```
Atom := Literal | StringTemplate | SelfReference | Enumeration | ParExpression
```

A *primary* is formed by recursively invoking or evaluating members of an atom or toplevel method or class.

```
Primary := Atom | Meta | EnumeratedInstanceReference | CallableReference | ValueReference | Invocation
```

More complex expressions are formed by combining expressions using operators, including assignment operators.

```
Expression := Primary | OperatorExpression
```

Parentheses are used for grouping:

```
ParExpression := "(" Expression ")"
```

Ceylon expressions are validated for typesafety at compile time. To determine whether an expression is assignable to a program element such as an attribute, local or formal parameter, Ceylon considers the *type* of the expression (the type of the objects that are produced when the expression is evaluated). An expression is assignable to a program element if the type of the expression is assignable to the declared type of the program element.

TODO: We need to allow some way to have an expression to be specified as a toplevel element, for DSLs.

TODO: Do we need a definition of "constant expression"? We might use it for:

- *`case` expressions (when are these evaluated?),*
- *annotations available at compile time,*
- *default parameter values (when are these evaluated?), and*
- *initializer/specifier expressions in second part of class body.*

For example, a constant expression might be anything formed from literals, enumerated instance references, metamodel references, the `..` and `->` operators and sequence enumerations.

5.1. Object instances, identity, and reference passing

An *object* is a unique identifier, together with a reference to a class, and a value for each simple attribute of the class (including inherited simple attributes). The object is said to be an *instance* of the class.

A *value* is a reference to an object (a copy of its unique identifier). At a particular point in the execution of the program,

every attribute of every object that exists, and every initialized local of every method or initializer that is currently executing has a value. Furthermore, every time an expression is executed, it produces a value.

Two values are said to be *identical* if they are references to the same object—if they hold the same unique identifier. The program may determine the if two values are identical using the `===` operator. It may not directly obtain the unique identifier (which is a purely abstract construct).

Invocation of a method or class initializer results in execution of the method with formal parameter values that are copies of the value produced by executing the argument expressions of the invocation, and a reference to the receiving instance that is a copy of the value produced by executing the receiver expression. The value produced by the invocation expression is a copy of the value produced by execution of the `return` directive expression.

```
Person myself(Person me) { return me }
Person p = ...;
assert() that (myself(p)===p); //assertion never fails
```

```
Semaphore s = Semaphore();
this.semaphore = s;
assert() that (semaphore===s) //assertion never fails
```

A new object is produced by execution of a class instantiation expression. The Ceylon compiler guarantees that if execution of a class initializer terminates with no uncaught exception, then every simple attribute of the object has been initialized. The value of an non-`mutable` simple attribute or local is initialized for the first time by execution of a specifier. Every class instantiation expression results in an object with a new unique identifier shared by no other existing object. The object exists from the point at which execution of its initializer terminates. *Conceptually*, the object exists until execution of the program terminates.

In practice, the object exists at least until the point at which it is not reachable by recursively following references from a local in a method or initializer currently being executed, or from an expression in a statement currently being executed. At this point, its simple attribute values are no longer accessible to expressions which subsequently execute and the object may be destroyed by the virtual machine. There is no way for the program to determine that an object has been destroyed by the virtual machine (unlike Java, Ceylon does not provide object finalizers).

A a special exception to the rules defined above:

- The compiler is permitted to emit bytecode that unexpectedly creates or avoids creating an actual instance of the erasable type `lang.Optional`, of `lang.Referenceable`, or of `lang.Assignable`, when an expression is evaluated, as long as this does not affect the execution of the program. (Therefore, the `===` operator is undefined for these types.)
- The compiler is permitted to emit bytecode that produces a new instance of one of the the built-in numeric types, of `lang.Range`, of `lang.Entry`, or of `lang.String` without execution of the initializer of the class, whenever any expression is evaluated. Furthermore, it is permitted to use such a newly-produced instance as the value of the expression, as long as the newly-produced instance is equal to the value expected according to the rules above, as determined using the `==` operator. Therefore, the `===` operator is unreliable for the listed types.

```
Boolean identical(Natural x, Natural y) { return x==y }
Natural n = 1;
assert() that (identical(n,n)); //assertion may fail!
```

```
String name = "Gavin";
person.name := name;
assert() that (person.name===name) //assertion may fail!
```

The execution of a Ceylon program complies with the rules laid out by the Java programming language's execution model (Chapter 17 of the Java Language Specification). Ceylon attributes and locals are considered *variables* in the sense of the JLS. Evaluation is considered a *use* operation, and assignment is considered an *assign* operation, again in terms of the JLS.

5.2. Literals

Ceylon supports literal values of the following types:

- `Natural` and `Float`,
- `Character`,

- `String`, and
- `Quoted`.

Ceylon does not need a special syntax for `Boolean` literal values, since `Boolean` is just a `Selector` with the enumerated instances `true` and `false`. The `null` value is just the enumerated instance of the class `Null`.

```
Literal := NaturalLiteral | FloatLiteral | CharacterLiteral | StringLiteral | QuotedLiteral
```

All literal values are instances of immutable types. The value of a literal expression is an instance of the type. How this instance is produced is not specified here.

5.2.1. Natural number literals

A natural number literal is an expression of type `lang.Natural`.

```
Natural m = n + 10;
```

Negative `Integer` values can be produced using the unary `-` operator:

```
Integer i = -1;
```

5.2.2. Floating point number literals

A floating point number literal is an expression of type `lang.Float`.

```
public Float pi = 3.14159;
```

5.2.3. Character literals

A single character literal is an expression of type `lang.Character`.

```
if ( string[i] == `+` ) { ... }
```

TODO: do we really need character literals?

5.2.4. Character string literals

A character string literal is an expression of type `lang.String`.

```
person.name := "Gavin King";
```

```
String multiline = "Strings may
span multiple lines
if you prefer.";
```

```
display("Melbourne\tVic\tAustralia\nAtlanta\tGA\tUSA\nGuanajuato\tGto\tMexico\n");
```

5.2.5. Single quoted literals

Single-quoted strings are used to express literal values for dates, times, regexes and hexadecimal numbers, and even for more domain-specific things like names, cron expressions, internet addresses, and phone numbers. This is an important facility since Ceylon is a language for expressing structured data.

A single quoted literal is an expression of type `lang.Quoted`. An extension is responsible for converting it to the appropriate type.

```
Date date = '25/03/2005';
```

```
Time time = '12:00 AM PST';
```

```
Boolean isEmail = '^\w+@((\w+)\.)+$'.matches(email);
```

```
Cron schedule = '0 0 23 ? * MON-FRI';
```

```
Color color = 'FF3B66';
```

```
Url url = 'http://jboss.org/ceylon';
```

```
mail.to:='gavin@hibernate.org';
```

```
PhoneNumber ph = '+1 (404) 129 3456';
```

```
Duration duration = '1h 30m';
```

Extensions that apply to the type `Quoted` are evaluated at compile time for single-quoted literals, alowing compile-time validation of the contents of the single-quoted string.

```
public extension Date date(Quoted dateString) { return ... }
```

```
public extension class Regex(Quoted expression) { return ... }
```

TODO: we should try to support interpolated expressions, just like we do for string literals.

TODO: Quoted literals are used for version numbers and version constraints in the module architecture, for example: `'1.2.3BETA'`.

5.3. String templates

A character string *template* contains interpolated expressions, surrounded by character string fragments.

```
StringTemplate := StringLiteral (Expression? StringLiteral)+
```

A character string template is an expression of type `StringTemplate`.

```
log.info("Hello, " person.firstName " " person.lastName ", the time is " Time() ".");
```

```
log.info("1 + 1 = " 1 + 1 "");
```

An interpolated expression in a string template may invoke or evaluate:

- any class member that is visible to the containing scope in which the literal appears, and
- any non-`mutable` local, block local attribute getter or block local method declared earlier within the containing scope.

An interpolated expression in a string template may not refer to `mutable` locals from the containing scope.

Interpolated expressions are evaluated when the `interpolate()` method of `StringTemplate` is called to produce a constant character string.

5.4. Self references

The type of the following special values depends upon the context in which they appear.

```
SelfReference := "this" | "super"
```

The keyword `super` refers to the current instance (the instance that is being invoked), and has the same members as the immediate superclass of the class, except for `fixed` members. Any invocation of this reference is processed by the method or attribute defined or inherited by this superclass, bypassing any method declaration that overrides the method on the current class or any subclass of the current class. Invocation of `fixed` members upon `super` is not allowed. The `super` reference is

not assignable to any type.

The keyword `this` refers to the current instance, and is assignable to both the type of the current class (the class which declares the method being invoked), and to the special type `subtype`, representing the concrete type of the current instance.

5.5. Metamodel references

The metamodel object representing a type or program element may be obtained using a completely typesafe syntax.

```
Meta := TypeMeta | MethodMeta | AttributeMeta | FunctionMeta
```

Metamodel references are compile-time typesafe.

```
Type<List<String>> stringListType = List<String>;
```

```
Class<Person,Name> personClass = Person;
```

```
Method<Log, Void, String> infoMethod = Log.info;
```

```
Attribute<Person, Name> nameAttribute = Person.name;
```

A metamodel reference that refers to a generic declaration must specify type arguments.

TODO: can we just default the type arguments to the upper bounds of the type parameters if the type arguments are missing? Or do we need some kind of `GenericType` object to represent generic declarations?

TODO: are there metamodel objects for block local declarations? This includes block local declarations inside a class body, and block local declarations inside a method or attribute body.

5.5.1. Interface and class metamodel references

A `Type` object may be obtained by specifying the full type, including type arguments if the type is generic.

```
TypeMeta := Type
```

The metamodel expression, `X`, for a type, class or interface is:

- of type `Interface<X>` where `X` is the interface, or
- assignable to `Class<X,P...>` where `X` is the toplevel class and `P...` are the types of the formal parameter list of the class, for every overloaded version of the class, or
- assignable to `MemberClass<X,Y,P...>` where `Y` is the member class and `P...` are the types of the formal parameter list of the class, for every overloaded version of the class.

Note that for a toplevel class `X`, the expression `X` is both a metamodel reference and a callable reference. This is consistent, since `Class<X,P...>` is a subtype of `Callable<X,P...>`.

5.5.2. Toplevel method metamodel references

A `Function` object representing a toplevel method may be obtained by specifying the method name, with type arguments if the toplevel method is generic.

```
FunctionMeta := MemberName TypeArguments?
```

The metamodel expression, `method`, for a toplevel method is:

- assignable to `Function<R,P...>` where `R` is the callable type of a method produced by removing the first formal parameter list of the method, and `P...` are the types of the first formal parameter lists of the method, for every overloaded version of the method.

Note that for a toplevel method `x()`, the expression `x` is both a metamodel reference and a callable reference. This is consistent, since `Function<X,P...>` is a subtype of `Callable<X,P...>`.

5.5.3. Member method metamodel references

A `Method` object representing a member method may be obtained by specifying the type (with type arguments) and member name, together with type arguments if the method is generic.

```
MethodMeta := Type "." MemberName TypeArguments?
```

The metamodel expression, `X.member`, for a member method is:

- assignable to `Method<X,R,P...>` where `X` is the type that defines the method, and `R` is the callable type of a method produced by removing the first formal parameter list of the method, and `P...` are the types of the first formal parameter lists of the method, for every overloaded version of the method.

5.5.4. Attribute metamodel references

An `Attribute` object representing an attribute may be obtained by specifying the type (with type arguments) and member name.

```
AttributeMeta := Type "." MemberName
```

The metamodel expression, `X.member`, for an attribute is:

- of type `Attribute<X,T>` where `X` is the type that defines the attribute, and `T` is the declared type of the attribute, unless the attribute is declared `mutable`, or
- of type `MutableAttribute<X,T>` where `X` is the type that defines the attribute, and `T` is the declared type of the attribute, if the attribute is declared `mutable`.

5.5.5. Using the metamodel

The metamodel object for a type allows its members to be iterated:

```
Type<Object> t = object.type;
for (Attribute<Object,String> a in t.attributes(String)) {
    log.info(a.name + "=" + a(object));
}
for (Method<Object,String> m in t.methods(Callable<String>)) {
    log.info(m.name + "=" + m(object)());
}
```

The metamodel object for a class, attribute or method implements `Callable` and is therefore invokable.

```
Class<ArrayList<String>,String[]> arrayListClass = ArrayList<String>;

List list = arrayListClass("foo", "bar", "baz);
```

```
Attribute<Person, String> nameAttribute = Person.name;

String personName = nameAttribute(person);
```

```
Method<Person, String> sayMethod = Person.say;

String result = sayMethod(person)();
```

```
MutableAttribute<Counter, Natural> countAttribute = Counter.count;

countAttribute(this)++;
```

The metamodel object for a class, attribute or method supports registration of a listener, which intercepts invocations.

```
MutableAttribute<Counter, Natural> countAttribute = Counter.count;
```

```
countAttribute.intercept()
    onGet (Counter c, Natural proceed()) {
        log.debug("getting");
        return proceed()
    };

countAttribute.intercept()
    onSet (Counter c, void proceed(Natural n), Natural value) {
        log.debug("setting");
        proceed(value)
    };
```

```
Method<Order,Item,Product,Natural> createItemMethod = Order.createItem;

createItemMethod.intercept()
    onInvoke (Order o, Item proceed(Product p, Natural n),
              Product product, Natural quantity) {
        log.debug("invoking in transaction");
        try (Transaction()) {
            return proceed(product,quantity)
        }
    };
```

TODO: According to this, we can "curry" in type arguments of the type. We need this. But if so, why can't we curry type arguments of the member? It's not a problem from the grammar point of view.

5.6. Enumerated instance references

An enumerated instance of a class is identified according to:

```
EnumeratedInstanceReference := (Type ".")? MemberName
```

The type is required unless the enumerated instance was explicitly imported or is a member of a containing class.

The type of the enumerated instance reference is the class of which the enumerated instance is a member.

The value of an enumerated instance reference is the instance of the class that was instantiated when the class was loaded by the virtual machine.

```
DayOfWeek sunday = DayOfWeek.sun;
```

5.7. Callable references

A *callable reference* is a reference to something—a method, callable parameter, or class—that can be *invoked* by specifying a list of arguments.

```
CallableReference := MethodReference | ConstructorReference
```

A callable reference may be invoked immediately, or it may be passed to other code which may invoke the reference. A callable reference captures the return type and formal parameter lists of the method, callable parameter, or class it refers to, allowing compile-time validation of argument types when the callable reference is invoked.

A callable reference expression is assignable to `Callable<T,P...>` where `T` and `P...` depend upon the schema of the method or class.

5.7.1. Receiver expressions

A callable reference or value reference may specify a *receiver expression*. The receiver expression produces the instance upon which a member is invoked or evaluated. The type of the receiver expression must have a member with the specified name.

```
Receiver := Primary
```

A receiver expression must be explicitly specified, unless:

- the reference is to a toplevel method or class,
- the reference is to a local or formal parameter, or
- the current instance of a containing class is the receiver.

When a callable reference with a receiver expression is executed, the receiver expresson is evaluated and a reference to the resulting value is held as part of the callable reference. When a value reference with a receiver expression is executed, the receiver expression is evaluated and a reference to the resulting value is held as part of the value reference.

5.7.2. Type arguments

If a callable reference expression refers to a generic declaration, either:

- it must be immediately followed by an argument list, allowing the compiler to infer the type arguments, or
- it must specify an explicit type argument list.

5.7.3. Method references

If a callable reference specifies a method name (or callable parameter name), it is called a *method reference*.

```
MethodReference := (Receiver ".")? MemberName TypeArguments?
```

The type of a method reference expression is assignable to the callable type of every overloaded version of the method. Calling the callable reference results in execution of the method.

5.7.4. Constructor references

If a callable reference specifies a class name, it is called a *constructor reference*.

```
ConstructorReference := (Receiver ".")? TypeName TypeArguments?
```

The type of a constructor reference expression is assignable to the callable type of every overloaded version of the class. Calling the constructor reference results in instantiation of the class.

5.7.5. Callable objects as method implementations

An expression of type `Callable` may be used to define a method using `=`. The expression type must be assignable to the callable type of the method being defined.

```
Comparison order(X x, Y y) = Order.reverse;
```

```
void display(String message) = log.info;
```

```
String newString(Character... chars) = String;
```

5.7.6. Callable objects as callable parameter arguments

An expression of type `Callable` may appear as an argument to a callable parameter, either as a positional argument, or as an argument specified using `=` in a named parameter invocation. The expression type must be assignable to the callable type of the callable parameter.

This method has a callable parameter:

```
void sort(List<String> list, Comparison by(String x, String y)) { ... }
```

This code passes a reference to a local method to the method.

```
Comparison reverseAlpha(String x, String y) { return y<=>x }
```

```
sort(names, reverseAlpha);
```

This class has two callable parameters:

```
public class TextInput(Natural size=30, String onInit(), String onUpdate(String s)) { ... }
```

This code instantiates the class, passing references to the `getIt()` and `setIt()` methods of the `Assignable` for the attribute `person.name`:

```
TextInput it = TextInput {
    size=15;
    onInit = person.name.getIt;
    onUpdate = person.name.setIt;
}
```

5.7.7. Callable objects as method return values

An expression of type `Callable` may be returned by a method with multiple parameter lists. The expression must be assignable to the callable type of a method formed by eliminating the first parameter list of the method.

```
Comparison getOrder(Boolean reverse=false)(Natural x, Natural y) {
    if (reverse) {
        Comparison reverse(Natural x, Natural y) { return y<=>x }
        return reverse
    else {
        Comparison natural(Natural x, Natural y) { return x<=>y }
        return natural
    }
}
```

This is slightly simpler using type inference:

```
Comparison getOrder(Boolean reverse=false)(Natural x, Natural y) {
    if (reverse) {
        local reverse(Natural x, Natural y) { return y<=>x }
        return reverse
    else {
        local natural(Natural x, Natural y) { return x<=>y }
        return natural
    }
}
```

Calling a method with multiple parameter lists is similar to the operation of "currying" in a functional programming language.

```
Comparison order(Natural x, Natural y) = getOrder();
Comparison comp = order(1,-1);
```

This is even simpler using type inference:

```
local order(Natural x, Natural y) = getOrder();
local comp = order(1,-1);
```

Or, if we don't need parameter names, we don't even need to explicitly declare the parameter list:

```
local order = getOrder();
local comp = order(1,-1);
```

In this case, `order` is actually not a method at all—it's just a local of type `Callable<Comparison,Natural,Natural>`.

Of course, more than one argument list may be specified in a single expression:

```
Comparison comp = getOrder(true)(10, 100);
```

5.8. Value references

A *value reference* is a reference to something—an attribute or local—that can be *evaluated*.

A value reference may be transparently converted to the actual current value of the attribute or local, or it may be passed, as a reference, to other code, which may invoke the reference to *evaluate* the current value of the attribute or local. A reference to a `mutable` attribute or local may be invoked to *assign* a new value to the attribute or local.

A value reference expression is assignable to `Referenceable<T>` or `Assignable<T>` were `T` is the type of the attribute or local.

5.8.1. Attribute and local references

An *attribute reference* is a reference to an attribute of some object. A *local reference* is a reference to a local or formal parameter. An attribute reference specifies the name of the attribute. A local reference specifies the name of the local or formal parameter.

```
ValueReference := (Receiver ".")? MemberName
```

The type of an attribute reference or local reference expression for an attribute, local, or formal parameter of declared type `X` is:

- `X`, if `X` is assignable to `Referenceable<Object>`, or, otherwise,
- `Referenceable<X>`, if the attribute, local, or formal parameter is not `mutable`, or if the setter is not visible to the block containing the attribute or local reference, or
- `Assignable<X>`, if the attribute, local, or formal parameter is `mutable` and the setter, if any, is visible to the block containing the attribute or local reference.

TODO: an alternative solution to the "recursion" problem would be to require an explicit call to `reference` defined by an intermediate interface to obtain the `Referenceable` instead of making it an extension, perhaps hiding the call behind a `&` operator. That would allow us to pass references to references. Yet another alternative is let the method of an extended type "underride" the method of the introduced type.

5.8.2. Pass by reference

An expression of type `Referenceable` or `Assignable` may be assigned to an attribute or local, passed as an argument to a method or initializer, or returned by a method. If the attribute, local, or formal parameter to which the expression is assigned is of declared type `Referenceable` or `Assignable`, or if the method which returns the expression is of declared type `Referenceable` or `Assignable`, this is called *pass by reference*, since it does not result in immediate evaluation of the referenced value. Instead, the attribute, local, or formal parameter holds a *reference* to the value of some other attribute, local, or formal parameter.

This local holds a reference:

```
Referenceable<String> nameRef = person.name;
```

This method returns a reference:

```
Assignable<String> getName(Person p) { return p.name }
```

This class has an initialization parameter which accepts a reference:

```
class Input(Assignable<String> model) { ... }
```

An attribute or local of type `Referenceable` or `Assignable` may not be declared `mutable`.

The `Referenceable` object representing a `mutable` local may not be assigned to an attribute, passed as a method argument, passed to a control directive, enumerated as the value of a sequence enumeration, or referred to by a nested method or class. It may be assigned to a local, or invoked directly inside the block that obtained it.

TODO: nail down these rules a little better.

5.9. Invocation

A callable object—any instance of `Callable`—is *invokable*. An *invocation* consists of an *invoked expression* of type `Callable<T,P...>`, together with an argument list and, optionally, an explicit type argument list.

```
Invocation := Primary Arguments
```

Any invocation expression where the invoked expression is a callable reference expression is called a *direct invocation* of the method, callable parameter, or class. In the case of a direct invocation, the compiler has additional information about the schema of the method or class that is not reified by the `Callable` interface. The compiler is aware of:

- all the overloaded versions of the method or class (the various `Callable` types to which the callable reference expression is assignable),
- the names of the formal parameters of the method or class,
- which formal parameters are defaulted, and their default values, and
- whether the last formal parameter in the list is a varargs parameter.

(Furthermore, in the case of a direct invocation, type argument inference is possible, since the compiler is aware of the type parameters and constraints of the method or class.)

An invocation must specify arguments for parameters of the callable object, either by listing parameter values in order or, in the case of a direct invocation, listing named parameter values.

```
Arguments := PositionalArguments FunctionalArguments? | NamedArguments
```

Arguments to required parameters must be specified. If the invocation is a direct invocation, arguments to defaulted parameters may optionally be specified, and one or more arguments to a varargs parameter may optionally be specified. Otherwise, an argument must be specified for each defaulted parameter, and a single argument must be specified for the varargs parameter.

For a required or defaulted formal parameter of type `T`, the type of the corresponding argument expression must be assignable to `T`.

For a tuple parameter of psuedo-type `P...`, the type of the corresponding argument expression must be `P...`. (That is, it must be a tuple parameter of declared type `P...`.)

For a varargs parameter of type `T...`, there may either:

- be a single argument expression of type assignable to `T[]`, or
- in the case of a direct invocation, an arbitrary number of argument expressions of type assignable to `T`.

In the second case, the argument expressions are evaluated and collected into an instance of `T[]` when the invocation is executed.

When a invocation expression of a callable reference is executed:

- first, the invoked expression is executed to obtain the callable object, then
- each argument is evaluated in turn in the calling context, then
- the actual member to be invoked is determined by considering the runtime type of the receiving instance and the static types of the arguments, and then
- execution of the calling context pauses while the body of the method or initializer is executed by the receiving instance with the argument values, then
- finally, when execution of the method or initializer ends without a thrown exception, execution of the calling context resumes.

When an invocation expression of any other invoked expression is executed, the `call()` method of `Callable` is invoked.

The type of an invocation expression is the type argument to the second type parameter of the expression type `Callable`

(the return type). Thus, the type of a method invocation is the return type of the method and the type of a class instantation is the class. The type of a `void` method invocation is `Void`.

5.9.1. Method invocation

A method invocation evaluates to the return value of the method, as specified by the `return` directive. The argument values are passed to the formal parameters of the method, and the body of the method is executed.

The actual value that invocation of a `void` method returns is not specified here. However, the `is T` operator always evaluates to `false` for any type `T` other than `Void` when applied to the return value of an invocation of a `void` method.

```
log.info("Hello world!")
```

```
log.info { message = "Hello world!"; }
```

```
printer.print { join = ", "; "Gavin", "Emmanuel", "Max", "Steve" }
```

```
printer.print { "Names: ", from (Person p in people) select (p.name) }
```

```
set person.name("Gavin")
```

```
get process.args()
```

```
amounts.sort() by (Float x, Float y) ( x<=>y );
```

```
people.each() perform (Person p) { log.info(p.name); }
```

```
hash(default, firstName, initial, lastName)
```

```
hash { algorithm=default; firstName, initial, lastName }
```

```
from (people) where (Person p) (p.age>18) select (Person p) (p.name)
```

```
iterate (map)
    perform (String name->Object value) {
        log.info("Entry: " name "->" value "");
    };
```

5.9.2. Class instantiation

Invocation of a constructor reference is called *instantiation* of the type. A class instantiation evaluates to a new instance of the class. The argument values are passed to the initialization parameters of the class, and the initializer is executed.

```
Map<String, Person>(entries)
```

```
Point { x=1.1; y=-2.3; }
```

```
ArrayList<String> { capacity=10; "gavin", "max", "emmanuel", "steve", "christian" }
```

```
Iterable<String> tokens = input.Tokens();
```

```
Panel {
    label = "Hello";
    Input i = Input(),
    Button {
        label = "Hello";
        action() {
            log.info(i.value);
        }
    }
}
```

5.9.3. Positional arguments

When arguments are listed, the arguments list is enclosed in parentheses.

```
PositionalArguments := "(" ( Expression ("," Expression)* )? ")"
```

Positional arguments must be listed in the same order as the corresponding formal parameters.

- First, an argument of each required parameters must be specified, in the order in which the required parameters were declared. There must be at least as many arguments as required formal parameters.
- Next, arguments of the first arbitrary number of defaulted parameters may be specified, in the order in which the defaulted parameters were declared. If there are fewer arguments than defaulted parameters, the remaining defaulted parameters are assigned their default values.
- Finally, if arguments to all defaulted parameters have been specified, and if the method declares a varargs parameter, an arbitrary number of arguments to the varargs parameter may be specified.

For example:

```
(getProduct(id), 1)
```

5.9.4. Named arguments

When arguments are named, the argument list is enclosed in braces.

```
NamedArguments := "{" NamedArgument* VarargArguments? "}"
```

Named arguments may be listed in a different order to the corresponding formal parameters.

Required and defaulted parameter arguments are specified by name. Varargs are specified by listing them, without specifying a name, at the end of the argument list.

A named argument either:

- specifies its value using `=`, and is terminated by a semicolon, or
- only for callable parameters, specifies the type or `local`, a formal parameter list and a block of code (an inline method declaration).

```
NamedArgument := SpecifiedNamedArgument | FunctionalNamedArgument
```

```
SpecifiedNamedArgument := ParameterName Specifier ";"
```

```
FunctionalNamedArgument := (InferableType | "void") ParameterName FormalParams Block
```

For example:

```
{
    product = getProduct(id);
    quantity = 1;
}
```

```
{
    label = "Say Hello";
    void onClick() {
        say("Hello");
    }
}
```

```
{
    Comparison by(X x, X y) { return x<=>y }
}
```

This is simpler using type inference:

```
{
    local by(X x, X y) { return x<=>y }
}
```

TODO: should the named parameter block be allowed to contain arbitrary statements? This is more regular, since you can do it in the body of a class, and attribute/method overriding is the model that we are following here. And it could be very useful when defining structured data.

5.9.5. Vararg arguments

Vararg arguments are seperated by commas.

```
VarargArguments := VarargArgument ("," VarargArgument)*
```

```
VarargArgument := Expression | InferableVariable Specifier
```

For example:

```
(1, 1, 2, 3, 5, 8)
```

A vararg argument may be a local declaration.

TODO: figure this out. Is this only for varargs in a named parameter invocation?

A vararg argument may be a `Sequence` of the parameter type.

```
( {1, 1, 2, 3, 5, 8} )
```

TODO: actually, upon reflection, I think it might be better and more regular to use semicolons to separate varargs in a named parameter invocation.

5.9.6. Default arguments

When no argument is assigned to a defaulted parameter by the caller, the default argument defined by the formal parameter declaration is used. The default argument expression is evaluated every time the method is invoked with no argument specified for the defaulted parameter.

This class:

```
public class Counter(Natural initialCount=0) { ... }
```

May be instantiated using any of the following:

```
Counter()
```

```
Counter(1)
```

```
Counter {}
```

```
Counter { initialCount=10; }
```

This method:

```
public class Counter() {

    package void init(Natural initialCount=0) {
        count:=initialCount;
    }

    ...

}
```

May be invoked using any of the following:

```
counter.init()
```

```
counter.init(1)
```

```
counter.init {}
```

```
counter.init { initialCount=10; }
```

5.9.7. Inline callable arguments

After a positional argument list, arguments to callable parameters may be specified with certain punctuation eliminated:

- the return type is not declared,
- if the callable parameter has an empty formal parameter list, the empty parentheses may be eliminated, and
- if the body of the method implementation consists of a single `return` directive followed by a parenthesized expression, the braces and `return` keyword may be eliminated.

These arguments are called an *inline callable arguments* of a positional parameter invocation.

```
FunctionalArguments := (ParameterName FunctionalBody)+
```

```
FunctionalBody := FormalParameters? ( Block | "(" Expression ")" )
```

For example:

```
where (Person p) (p.age>18)
```

```
by (Float x, Float y) ( x<=>y )
```

```
ifTrue (x+1)
```

```
each { count+=1; }
```

```
perform (Person p) { log.info(p.name); }
```

Inline callable arguments are listed without any additional punctuation:

```
ifTrue (x+1) ifFalse (x-1)
```

```
select (Person p) (p.name) where (Person p) (p.age>18)
```

Arguments must be listed in the same order as the formal parameters are declared by the method declaration.

TODO: should we support type inference for the formal parameters? It gets complicated with method overloading. For example:

```
by (local x, local y) ( x<=>y )
```

```
where (local p) (p.age>18)
```

5.9.8. Iteration

A specialized invocation syntax is provided for toplevel methods which iterate collections. If the first parameter of the method is of type `Iterable<X>`, annotated `iterated`, and all remaining parameters are callable parameters, then the method may be invoked according to the following protocol:

```
Primary "(" ForIterator ")" FunctionalArguments
```

And then if a callable parameter has a formal parameter of type `X` annotated `coordinated`, that parameter need not be declared by its argument. Instead, the parameter is declared by the iterator.

For example, for the following method declaration:

```
public
List<Y> from<X,Y>(iterated Iterable<X> elements,
                  Boolean where(coordinated X x),
                  Y select(coordinated X x));
```

We may invoke the method as follows:

```
List<String> names = from (Person p in people) where (p.age>18) select (p.name);
```

Which is equivalent to:

```
List<String> names = from (people) where (Person p) (p.age>18) select (Person p) (p.name);
```

Or, we may invoke the method as follows:

```
List<String> labels = from (Key key -> Value value in namedValues)
                      where (user.authorized(key))
                      select ($key + ": " + $value);
```

Which is equivalent to:

```
List<String> labels = from (namedValues)
                      where (Key key -> Value value) (user.authorized(key))
                      select (Key key -> Value value) ($key + ": " + $value);
```

Type inference simplifies this further:

```
local names = from (local p in people) where (p.age>18) select (p.name);
```

5.9.9. Variable definition

A specialized invocation syntax is also provided for toplevel methods which define a variable. If the first parameter of the method is of type `X`, annotated `specified`, and all remaining parameters are callable parameters, then the method may be invoked according to the following protocol:

```
Primary "(" InferableVariable Specifier ")" FunctionalArguments
```

And then if a callable parameter has a formal parameter of type `X` annotated `coordinated`, that parameter need not be declared by its argument. Instead, the parameter is declared by the variable specifier.

For example, for the following method declaration:

```
public
Y ifExists<X,Y>(specified X? value,
                Y then(coordinated X x),
                Y otherwise());
```

We may invoke the method as follows:

```
Decimal amount = ifExists(Payment p = order.payment) then (p.amount) otherwise (0.0);
```

Which is equivalent to:

```
Decimal amount = ifExists(order.payment) then (Payment p) (p.amount) otherwise (0.0);
```

And for the following method declaration:

```
public
```

```
Y using<X,Y>(specified X resource,
             Y seek(coordinated X x))
  given X satisfies Usable;
```

We may invoke the method as follows:

```
Order order = using (Session s = Session()) seek (s.get(Order, oid));
```

Which is equivalent to:

```
Order order = using (Session()) seek (Session s) (s.get(Order, oid));
```

Type inference simplifies this further:

```
local amount = ifExists(local p = order.payment) then (p.amount) otherwise (0.0);
```

```
local order = using (local s = Session()) seek (s.get(Order, oid));
```

5.9.10. Resolving direct invocations of overloaded methods and classes

A direct invocation is resolved to a specific toplevel declaration or member of the receiving type at compile time, even if the method or class it refers to is overloaded.

The *initial signature* of a method or class in a direct invocation is formed by:

- taking the signature of the formal parameter list of the member class, or of the first formal parameter list of the method, and
- replacing each occurrence of any type parameter of the receiving type in the signature with the type argument of that parameter in the callable object expression type, and
- replacing each occurrence of any type parameter of the method or class in the signature with the explicitly specified type argument, if type arguments were specified, or with the first declared upper bound of the type parameter, or `lang.Void` if the type parameter has no declared upper bound.

A direct invocation is *resolveable* if there is exactly one method, callable parameter, or class declaration which:

- has the specified name,
- has the same number of parameters as specified argument expressions,
- has a type parameter list to which the type argument list conforms, if type arguments were explicitly specified,
- specifies generic type constraints which are satisfied by the type arguments, if type arguments were explicitly specified, and which
- has an initial signature to which the given argument expression types are assignable.

In the case of a method with multiple formal parameter lists, only the first formal parameter list is considered.

If more than one overloaded declaration has an initial signature to which the arguments are assignable, or if there is no declaration with an initial signature to which the arguments are assignable, the invocation or instantiation is illegal. (Note that Ceylon is stricter and simpler than Java with this rule.)

Finally, if type arguments were not explicitly specified, there must be a combination of type arguments that can be substituted for the type parameters of the method or type, respecting constraints upon the type parameters, that results in a method schema such that:

- the given argument expression types are assignable to the method parameter types after substitution of the type arguments, and
- the expression type of the invocation or instantiation, after substitution of the type arguments, is assignable to the sur-

rounding context.

TODO: Figure out the details of the type inference implied by this last bit!

If no such combination of type arguments exists, the invocation or instantiation is illegal. (Note that Ceylon is less strict than Java with this rule.)

5.10. Evaluation and assignment

A reference object—any instance of `Referenceable`—can be *evaluated* to produce a value. If the reference object is of type `Assignable`, its value may also be *assigned*. It is not necessary to explicitly invoke the reference object to evaluate or assign the referenced value. Instead:

- the `getIt()` method of `Referenceable` is an `extension`, and transparently produces the current value of a referenced value without any explicit invocation, and
- the `:=` operator is defined to apply directly to argument expressions of type `Assignable`, hiding the underlying invocation of the `setIt()` method.

Thus, the following are equivalent:

```
String name = person.name.getIt();   //discouraged
```

```
String name = person.name;           //preferred
```

As are the following:

```
person.name.setIt("Gavin");   //discouraged
```

```
person.name := "Gavin";       //preferred
```

Furthermore, the argument specification operators cannot be applied directly to a value reference, and so a value reference is not a callable reference. However, it is possible to obtain a callable reference to:

- the `getIt()` method of `Referenceable`, or
- the `setIt()` method of `Assignable`.

These methods, conceptually, do the work of evaluating or assigning a value reference.

5.10.1. Evaluation

Invocation of the `getIt()` method of an attribute reference or local reference evaluates the attribute or local. The `getIt()` method is declared `extension`, so it does not need to be called explicitly.

```
String name = person.name;
```

Sometimes we need to pass an attribute by reference:

```
Referenceable<String> nameRef = person.name;
String name = nameRef;
```

Sometimes we need to transform an attribute reference into a method reference:

```
String getName() = person.name.getIt;
String name = getName();
```

When a local evaluation is executed, the current value of the local is immediately obtained. The resulting value is the current value of the local.

When an attribute evaluation is executed:

- first, the reference expression is executed to obtain the reference object, then
- the actual member to be invoked is determined by considering the runtime type of the receiving instance, and then
- if the member is a simple attribute, the current value of the simple attribute is retrieved from the recieving instance, or
- otherwise, execution of the calling context pauses while the body of the attribute getter is executed by the receiving instance, then,
- finally, when execution of the getter ends without a thrown exception, execution of the calling context resumes.

The resulting value is the current value of the simple attribute or the return value of the attribute getter, as specified by the `return` directive.

Note that the `value()` method of `Correspondence` is also defined to return a `Referenceable`.

```
Person person = people[index];
```

Therefore an element expression can be passed by reference:

```
Referenceable<Person> personRef = person[index];
Person person = person;
```

And an element expression can be transformed into method reference:

```
Person getPerson() = people.[index].getIt;
Person person = getPerson();
```

5.10.2. Assignment

Invocation of the `setIt()` method of an attribute reference of type `Assignable` assigns of the attribute or local. We usually use an assignment operator instead of invoking `setIt()` directly.

```
person.name := "Gavin";
```

Sometimes we need to pass a mutable attribute by reference:

```
Assignable<String> nameRef = person.name;
nameRef := "Gavin";
```

Sometimes we need to transform a mutable attribute reference into a method reference:

```
String setName(String name) = person.name.setIt;
setName("Gavin");
```

Even the following are possible:

```
Assignable<String> getName(Person p) { return p.name }
getName(p) := "Gavin";
```

```
 setName(Assignable<String> name, String value) { name:=value; }
setName(p.name, "Gavin");
```

When an attribute value is assigned:

- first, the reference expression is executed to obtain the reference object, then
- the actual member to be invoked is determined by considering the runtime type of the receiving instance, and then
- if the member is a simple attribute or local, the value of the simple attribute of local is set to the new value,
- otherwise, execution of the calling context pauses while the body of the attribute setter is executed by the receiving instance with the new value, then,

- finally, when execution of the setter ends without a thrown exception, execution of the calling context resumes.

Note that the `value()` method of `OpenCorrespondence` is also defined to return an `Assignable`.

```
people[index] := person;
```

Therefore an assignable element expression can be passed by reference:

```
Assignable<Person> personRef = people[index];
personRef := person;
```

And an assignable element expression can be transformed into method reference:

```
Person setPerson(Person p) = people[index].setIt;
setPerson(person);
```

5.11. Enumeration

A sequence may be instantiated by enumerating the elements inside braces:

```
Enumeration := "{" ( Expression ("," Expression)* )? "}"
```

Each enumerated element expression type must be assignable to `Object`.

The value of an enumeration is a new instance of `Sequence`, containing the enumerated elements in the given order. When an enumeration is executed, each element expression is evaluated, and the resulting values collected together into an object that implements `Sequence<T>` where `T` is a subtype of `Object` to which all the enumerated element expression types are assignable. The concrete type of this object is not specified here.

```
String[] names = { "gavin", "max", "emmanuel", "steve", "christian" };
```

Empty braces `{}` are an abbreviation of `None.none`.

```
OpenList<Connection> connections = {};
```

There is no special syntax for constructing lists, sets or maps. However, the `..` and `->` operators, together with the convenient sequence enumeration syntax, and some built-in extensions help us achieve the desired effect.

```
List<String> languages = { "Java", "Ceylon", "Smalltalk", "C#" };
```

```
List<Natural> numbers = 1..10;
```

Sequences are transparently converted to sets or maps, allowing sets and maps to be initialized as follows:

```
Map<String, String> map = { "Java"->"Boring...", "Scala"->"Difficult :-(", "Ceylon"->"Fun!" };
```

```
Set<String> set = { "Java", "Ceylon", "Scala" };
```

```
OpenList<String> list = {};
```

TODO: an alternative to this syntax would be to allow a comma-separated list of values without braces after `=`, `in` or `return` and after `:=` in an attribute initializer. The disadvantage to this approach is that enumerations would not be allowed in positional parameter invocations, or in expressions. It's also less regular. On the other hand, it is consistent with how we treat varargs.

```
Map<String, String> map = "Java"->"Boring...", "Scala"->"Difficult :-(", "Ceylon"->"Fun!";
```

```
for (String lang in "Java", "Ceylon", "Scala", "C#") { ... }
```

```
join { sep=", "; strings=firstName, initial, lastName; };
```

5.12. Operators

Operators are syntactic shorthand for more complex expressions involving method invocation or attribute access. Each operator is defined for a particular type. There is no support for user-defined operator *overloading*. However, the semantics of an operator may be customized by the implementation of the type that the operator applies to. This is called *operator polymorphism*.

Some examples:

```
Float z = x * y + 1.0;
```

```
even := n % 2 == 0;
```

```
++count;
```

```
Integer j = i++;
```

```
if ( x > 100 || x < 0 ) { ... }
```

```
User? gavin = users["Gavin"];
```

```
List<Item> firstPage = list[0..20];
```

```
for ( Natural n in 1..10 ) { ... }
```

```
if (char in `A`..`Z`) { ... }
```

```
List<Day> nonworkDays = days[{0,7}];
```

```
Natural lastIndex = getLastIndex() ? sequence.lastIndex;
```

```
if ( exists name => name == "Gavin" ) { return ... }
```

```
log.info( "Hello " + $person + "!")
```

```
List<String> names = { person1, person2, person3 }[].name;
```

```
String? name = person?.name;
```

```
this.total += item.price;
```

```
if ( nonempty args[i] && !args[i].first == `-` ) { ... }
```

```
Float vol = length**3;
```

```
map.define(person.name->person);
```

```
order.lineItems[index] := LineItem { product = prod; quantity = 1; };
```

In all operator expressions, the arguments of the operator must be evaluated from left to right when the expression is executed. In certain cases, depending upon the definition of the operator, evaluation of the leftmost argument expression results in a value that causes the final value of the operator expression to be produced immediately without evaluation of the remaining argument expressions. Optimizations performed by the Ceylon compiler must not alter these behaviours.

5.12.1. Operator precedence

This table defines the relative precedence of the various operators, from highest to lowest, along with associativity rules:

Table 5.1.

Operations	Operators	Type	Associativity
Member invocation and lookup, subrange, postfix increment, postfix decrement:	`.`, `[].`, `?.`, `()`, `{}`, `[]`, `?[]`, `[..]`, `[...]`, `++`, `--`	Binary / ternary / N-ary / unary postfix	Left
Prefix increment, prefix decrement, negation, render, bitwise complement:	`++`, `--`, `-`, `$`, `~`	Unary prefix	Right
Exponentiation:	`**`	Binary	Right
Multiplication, division, remainder, bitwise and:	`*`, `/`, `%`, `&`	Binary	Left
Addition, subtraction, bitwise or, bitwise xor, list concatenation:	`+`, `-`, `|`, `^`	Binary	Left
Range and entry construction:	`..`, `->`	Binary	None
Existence, emptiness:	`exists`, `nonempty`	Unary postfix	None
Default:	`?`	Binary	Right
Comparison, containment, assignability:	`<=>`, `<`, `>`, `<=`, `>=`, `in`, `is`	Binary	None
Equality:	`==`, `!=`, `===`	Binary	None
Logical not:	`!`	Unary prefix	Right
Logical and:	`&&`	Binary	Left
Logical or:	`||`	Binary	Left
Logical implication:	`=>`	Binary	None
Assignment:	`:=`, `.=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `&&=`, `||=`, `?=`	Binary	Right

TODO: should `?` have a higher precedence?

TODO: should `^` have a higher precedence than `|` like in C and Java?

TODO: should `^`, `|`, `&` have a lower precedence than `+`, `-`, `*`, `/` like in Ruby?

Note: if we decide to add `<<` and `>>` later, we could give them the same precedence as `**`.

5.12.2. Operator definition

The following tables define the semantics of the Ceylon operators. There are four basic operators which do not have a definition in terms of other operators or invocations:

- the *member selection* operator `.` separates the receiver expression and member name in a callable reference expression or attribute reference expression,
- the *argument specification* operators `()` and `{}` specify the argument list of an invocation,
- the *identity* operator `===` evaluates to `true` if its argument expressions evaluate to references to the same object, and `false` otherwise, and
- the *assignability* operator `is` evaluates to `true` if its argument expression evaluates to an instance of a class assignable to the specified type, and `false` otherwise.

The argument expressions of the identity operator must be of type `lang.Object`. The argument expression of the assignability operator may be of any type, not limited to types assignable to `Object`.

All other operators are defined in terms of other operators and/or invocations.

In the tables, the following pseudo-code is used, which is not legal Ceylon syntax:

First,

```
if (b) then x else y   //pseudocode
```

means the same value produced by the Ceylon library method `ifTrue()`:

```
ifTrue (b) then (x) else (y)
```

Second,

```
if (exists o) then x else y   //pseudocode
```

means the same value produced by the Ceylon library method `ifExists()`:

```
ifExists (o) then (x) else (y)
```

Third,

```
for (X x in c) e      //pseudocode
```

means the same value produced by the Ceylon library method `from()`:

```
from (X x in c) select (e)
```

The tables define semantics only. The compiler is permitted to emit equivalent bytecode that produces the same value as the pseudo-code that defines the operator, without actually executing any invocation, for the following operators:

- all arithmetic operators,
- all bitwise operators,
- the comparison and equality operators `=`, `!=`, `<=>`, `<`, `>`, `<=`, `>=` when the argument expression types are built-in numeric types,
- the `Range` and `Entry` construction operators `..` and `->`,
- the sequence concatenation operator `+`, and
- all assignment and compound assignment operators.

Therefore, listeners registered for the method invocations and class instantiations that define these operators may not be called when the operator expressions are executed.

5.12.3. Basic invocation and assignment operators

These operators support method invocation and attribute evaluation and assignment. The `$` operator is a shortcut for converting any expression to a `String`.

Table 5.2.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Invocation						
`.`	`lhs.member`	member		`X`	`Member<X,T>`	`Referenceable<T>` or `T`

Op	Example	Name	Definition	LHS type	RHS type	Return type
`()` or `{}`	`lhs(x,y,z)` or `lhs{a=x;b=y;}`	invoke		`Callable<T,P...>`	`P...`	`T`
Assignment						
`:=`	`lhs := rhs`	assign	`lhs.setIt(rhs)`	`Assignable<X>`	`X`	`X`
Render						
`$`	`$rhs`	render	`this.string(rhs)`		`Object?`	`String`
Compound invocation assignment						
`.=`	`lhs.=member`	follow	`lhs:=lhs.member`	`X`	`Attribute<X,X>`	`X`
`.=`	`lhs.=member(x,y,z)`	apply	`lhs:=lhs.member(x,y,z)`	`X`	`Method<X,X,P...>`, together with arguments `P...`	`X`

TODO: we could perhaps move the member selection operator `.` down to `Object` from `Void` if we somehow turned `getIt()` and `setIt()` into basic operators. Not sure if this would be advantageous.

TODO: do we really need the `$` operator?

5.12.4. Equality and comparison operators

These operators compare values for equality, order, magnitude, or membership, producing boolean values.

Table 5.3.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Equality						
`===`	`lhs === rhs`	identical		`Object`	`Object`	`Boolean`
`==`	`lhs == rhs`	equal	`if (exists lhs) lhs.equals(rhs) else if (exists rhs) false else true`	`Object?`	`Object?`	`Boolean`
`!=`	`lhs != rhs`	not equal	`if (exists lhs) lhs.equals(rhs).complement else if (exists rhs) true else false`	`Object?`	`Object?`	`Boolean`
Comparison						
`<=>`	`lhs <=> rhs`	compare	`lhs.compare(rhs)`	`Comparable<T>`	`T`	`Comparison`
`<`	`lhs < rhs`	smaller	`(lhs<=>rhs).smaller`	`Comparable<T>`	`T`	`Boolean`
`>`	`lhs > rhs`	larger	`(lhs<=>rhs).larger`	`Comparable<T>`	`T`	`Boolean`
`<=`	`lhs <= rhs`	small as	`(lhs<=>rhs).smallAs`	`Comparable<T>`	`T`	`Boolean`

Op	Example	Name	Definition	LHS type	RHS type	Return type
>=	lhs >= rhs	large as	(lhs<=>rhs).largeAs	Comparable<T>	T	Boolean
Containment						
in	lhs in rhs	in	lhs.in(rhs)	Object	Category or Iterable<Object>	Boolean
Assignability						
is	lhs is Rhs	is		X		Boolean

TODO: should we really have the equality operators accept null values?

5.12.5. Logical operators

These are the usual logical operations for boolean values.

Table 5.4.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Logical operations						
!	!rhs	not	if (rhs) false else true		Boolean	Boolean
||	lhs || rhs	conditional or	if (lhs) true else rhs	Boolean	Boolean	Boolean
&&	lhs && rhs	conditional and	if (lhs) rhs else false	Boolean	Boolean	Boolean
=>	lhs => rhs	implication	if (lhs) rhs else true	Boolean	Boolean	Boolean
Logical assignment						
||=	lhs ||= rhs	conditional or	if (lhs) true else lhs:=rhs	Assignable<Boolean>	Boolean	Boolean
&&=	lhs &&= rhs	conditional and	if (lhs) lhs:=rhs else false	Assignable<Boolean>	Boolean	Boolean

5.12.6. Operators for handling null values

These operators make it easy to work with `Optional` values.

Table 5.5.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Existence						
exists	lhs exists	exists	lhs is Something<Object>	Object?		Boolean
nonem	lhs nonempty	nonempty	if (exists lhs) !lhs.empty	Container?		Boolean

Op	Example	Name	Definition	LHS type	RHS type	Return type
`pty`			`else false`			
Default						
`?`	`lhs ? rhs`	default	`if (exists lhs) lhs else rhs`	`T?`	`T` or `T?`	`T` or `T?`
Default assignment						
`?=`	`lhs ?= rhs`	default assignment	`if (exists lhs) lhs else lhs:=rhs`	`Assignable<T?>`	`T` or `T?`	`T` or `T?`
Nullafe invocation						
`?.`	`lhs?.member`	nullsafe member	`if (exists lhs) lhs.member else null`	`X?`	`Member<X,T>`	`T?`
`()` or `{}`	`lhs(x,y,z)` or `lhs{a=x;b=y;}`	nullsafe invoke	`if (exists lhs) lhs(x,y,z) else null`	`Callable<T,P...>?`	`P...`	`T?`

5.12.7. Correspondence and sequence operators

These operators provide a simplified syntax for accessing values of a `Correspondence`, and for joining and obtaining subranges of `Sequencess`.

Table 5.6.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Keyed element access						
`[]`	`lhs[index]`	lookup	`lhs.value(index)`	`Correspondence<X,Y>`	`X`	`Y?`
`?[]`	`lhs?[index]`	nullsafe lookup	`if (exists lhs) lhs[index] else null`	`Correspondence<X,Y>?`	`X`	`Y?`
`[]`	`lhs[indices]`	sequenced lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`X[]`	`Y[]`
`[]`	`lhs[indices]`	iterated lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`Iterable<X>`	`Iterable<Y>`
Sequence subranges						
`[..]`	`lhs[x..y]`	subrange	`range(lhs,x,y)`	`X[]`	Two `Natural` values	`X[]`
`[...]`	`lhs[x...]`	upper range	`range(lhs,x)`	`X[]`	`Natural`	`X[]`
Sequence concatenation						
`+`	`lhs + rhs`	join	`join(lhs, rhs)`	`X[]`	`X[]`	`X[]`
Spread invocation						
`[].`	`lhs[].member`	spread member	`for (X x in lhs) x.member`	`X[]`	`Member<X,T>`	`T[]`
`()` or `{}`	`lhs(x,y,z)` or `lhs{a=x;b=y;}`	spread invoke	`for (C c in lhs) c(x,y,z)`	`Callable<T,P...>[]`	`P...`	`T[]`

TODO: Should we have operators for set union/intersection/complement and set comparison?

5.12.8. Operators for constructing objects

These operators simplify the syntax for constructing certain commonly used built-in types.

Table 5.7.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Range and entry constructors						
`..`	`lhs .. rhs`	range	`Range(lhs, rhs)`	`T given T satisfies Ordinal, Comparable<T>`	`T`	`Range<T>`
`->`	`lhs -> rhs`	entry	`Entry(lhs, rhs)`	`U`	`V`	`Entry<U,V>`

TODO: Should we have operators for performing arithmetic with datetimes and durations, constructing intervals and combining dates and times?

5.12.9. Arithmetic operators

These are the usual mathematical operations for all kinds of numeric values.

Table 5.8.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Increment, decrement						
`++`	`++rhs`	successor	`rhs:=rhs.successor`		`Assignable<Ordinal<T>>`	`T`
`--`	`--rhs`	predecessor	`rhs:=rhs.predecessor`		`Assignable<Ordinal<T>>`	`T`
`++`	`lhs++`	increment	`(++lhs).predecessor`	`Assignable<Ordinal<T>>`		`T`
`--`	`lhs--`	decrement	`(--lhs).successor`	`Assignable<Ordinal<T>>`		`T`
Numeric operations						
`-`	`-rhs`	negation	`rhs.inverse`		`Invertable<I>`	`I`
`+`	`lhs + rhs`	sum	`lhs.plus(rhs)`	`Numeric<N>`	`N`	`N`
`-`	`lhs - rhs`	difference	`lhs.minus(rhs)`	`Numeric<N>`	`N`	`N`
`*`	`lhs * rhs`	product	`lhs.times(rhs)`	`Numeric<N>`	`N`	`N`
`/`	`lhs / rhs`	quotient	`lhs.divided(rhs)`	`Numeric<N>`	`N`	`N`

Op	Example	Name	Definition	LHS type	RHS type	Return type
`%`	`lhs % rhs`	remainder	`lhs.remainder(rhs)`	`Integral<N>`	`N`	`N`
`**`	`lhs ** rhs`	power	`lhs.power(rhs)`	`Numeric<N>`	`N`	`N`
Numeric assignment						
`+=`	`lhs += rhs`	add	`lhs:=lhs+rhs`	`Assignable<Numeric<N>>`	`N`	`N`
`-=`	`lhs -= rhs`	subtract	`lhs:=lhs-rhs`	`Assignable<Numeric<N>>`	`N`	`N`
`*=`	`lhs *= rhs`	multiply	`lhs:=lhs*rhs`	`Assignable<Numeric<N>>`	`N`	`N`
`/=`	`lhs /= rhs`	divide	`lhs:=lhs/rhs`	`Assignable<Numeric<N>>`	`N`	`N`
`%=`	`lhs %= rhs`	remainder	`lhs:=lhs%rhs`	`Assignable<Integral<N>>`	`N`	`N`

Built-in converters allow for type promotion of numeric values used in expressions. Coverters exist for the following numeric types:

- `lang.Natural` to `lang.Integer`, `lang.Float`, `lang.Whole` and `lang.Decimal`
- `lang.Integer` to `lang.Float`, `lang.Whole` and `lang.Decimal`
- `lang.Float` to `lang.Decimal`
- `lang.Whole` to `lang.Decimal`

This means that `x + y` is defined for any combination of numeric types `X` and `Y`, except for the combination `Float` and `Whole`, and that `x + y` always produces the same value, with the same type, as `y + x`.

5.12.10. Bitwise operators

These are C-style bitwise operations for bit strings (unsigned integers). A `Boolean` is considered a bit string of length one, so these operators also apply to `Boolean` values. Note that in Ceylon these operators have a higher precedence than they have in C or Java. There are no bitshift operators in Ceylon.

Table 5.9.

Op	Example	Name	Definition	LHS type	RHS type	Return type
Bitwise operations						
`~`	`~rhs`	complement	`rhs.complement`		`Bits<X>`	`X`
`|`	`lhs | rhs`	or	`lhs.or(rhs)`	`Bits<X>`	`X`	`X`
`&`	`lhs & rhs`	and	`lhs.and(rhs)`	`Bits<X>`	`X`	`X`
`^`	`lhs ^ rhs`	exclusive or	`lhs.xor(rhs)`	`Bits<X>`	`X`	`X`
Bitwise assignment						

Op	Example	Name	Definition	LHS type	RHS type	Return type
`|=`	`lhs |= rhs`	or	`lhs:=lhs|rhs`	`Assignable<Bits<X>>`	`X`	`X`
`&=`	`lhs &= rhs`	and	`lhs:=lhs&rhs`	`Assignable<Bits<X>>`	`X`	`X`
`^=`	`lhs ^= rhs`	exclusive or	`lhs:=lhs^rhs`	`Assignable<Bits<X>>`	`X`	`X`

Chapter 6. Basic types

There are no primitive types in Ceylon, however, there are certain important types provided by the package `lang`. Many of these types support *operators*.

6.1. The void type

The class `lang.Void` is the root of the type system. All types are assignable to `Void`, which supports the the operator `.` (member selection), along with `is` (assignability).

```
public abstract class Void {}
```

6.2. The Object type

User-written classes extend `lang.Object` and all interface types are assignable to `lang.Object`. Expressions of type `Object` support the binary operator `===` (identity equals). An `Object` may belong to collections. Expressions of type `Object` therefore support the binary operator `in` (in).

```
public abstract class Object()
        extends Void() {

    doc "The equals operator |x == y|. Default implementation compares
         attributes annotated |id|, or performs identity comparison."
    see (id)
    public default Boolean equals(Object that) { return ... }

    doc "Compares the given attributes of this instance with the given
         attributes of the given instance."
    public Boolean equals(Object that, Attribute... attributes) { ... }

    doc "The hash code of the instance. Default implementation compares
         attributes annotated |id|, or assumes identity equality."
    see (id)
    public default Integer hash { return ... }

    doc "Computes the hash code of the instance using the given
         attributes."
    public Integer hash(Attribute... attributes) { ... }

    doc "A developer-friendly string representing the instance.
         By default, the string contains the name of the type,
         and the values of all attributes annotated |id|."
    public default String string { return ... }

    doc "A developer-friendly string representing the instance,
         containing the name of the type, and the value of the
         given attributes."
    public default String string(Attribute... attributes) { ... }

    doc "Determine if the instance belongs to the given |Category|."
    see (Category)
    public Boolean in(Category cat) {
        return cat.contains(this)
    }

    doc "Determine if the instance is one of the given objects."
    public Boolean in(Object... objects) {
        return forAny (Object elem in objects) some elem == this
    }

    doc "The |Type| of the instance."
    public Type<subtype> type { return ... }

    doc "A log obect for the type."
    public default Log log { return type.log }

    doc "Transform the given object to a string. Override to
         customize the render operator and character string
         template expression interpolation."
    public default String string(Object? object) {
        return (object ? nullString).string
    }

    doc "The string representation of a null value. Override
         to customize the render operator and character string
         template expression interpolation."
```

```
    public default String nullString = "null";

    ...

}
```

6.3. Callable references

The type `lang.Callable` represents an executable operation. Expressions of type `Callable` may be the subject of the argument specification operators `()` and `{}`.

```
public interface Callable<out R, P...> {
    public R call(P... args);
    public void intercept( R onInvoke(R proceed(P... args), P... args) )();
}
```

An interceptor may be removed by invoking the returned method reference:

```
void removeSayInterceptor() = person.say.intercept()
    onInvoke(void proceed(String words), String words) {
        proceed(words.uppercase);
    };
...
removeSayInterceptor();
```

The following extension provides function *composition*, combining a `Callable<Y,X>` with a `Callable<X,P...>` to produce a `Callable<Y,P...>`:

```
public extension Y composeable<X,Y,P...>(Y f(X x))(X g(P... args))(P... args) {
    Y compose(X g(P... args))(P... args) {
        Y composition(P... args) {
            return f(g(args))
        }
        return composition
    }
    return compose
}
```

For example:

```
void logFormatted(Date date) = log.info(DateFormat('dd/mm/yyyy').format);
```

The next extension allows the first parameter of any function with multiple parameters to be *curried*, transforming a `Callable<R,T,P...>` into a `Callable<Callable<R,P...>,T>`:

```
public extension class Curryable<R,T,P...>(R f(T t, P... args)) {
    R partial(T t)(P... p) {
        R curried(P... p) {
            return f(t, args)
        }
        return curried
    }
}
```

For example:

```
void info(String message) = log.message.partial(Level.info);
void hello() = info.partial("Hello!");
```

This extension does the reverse, transforming a `Callable<Callable<R,P...>,T>` into a `Callable<R,T,P...>`:

```
public extension class Flattenable<R,T,P...>(R f(T t)(P... args)) {
    R flatten(T t, P... p) {
        return f(t)(p)
    }
}
```

For example:

```
void say(Person p, String words) = Person.say.flatten;
```

Finally, this extension swaps the first and second formal parameter lists of a function with multiple parameter lists, transforming a `Callable<Callable<R,Q...>,P...>` into a `Callable<Callable<R,P...>,Q...>`:

```
public extension class Shuffleable<R,P...,Q...>(R f(P... p)(Q... q)) {
   R shuffle(Q... q)(P... p) {
       R shuffled(P... p) {
           return f(p)(q)
       }
       return shuffled
   }
}
```

For example:

```
Float sqr(Float x) = Float.power.shuffle(2);
void say(String words, Person p) = Person.say.shuffle.flatten;
```

6.4. Referenceable and assignable values

The class `lang.Referenceable` represents a value for which a reference can be obtained, allowing pass by reference semantics. Instances of `Referenceable<T>` are transparently assignable to `T`, according to the `extension` method `getIt()`.

```
public abstract class Referenceable<out T>
        extends Void() {

    doc "Obtain and return the value."
    extension T getIt();

    public void intercept( T onGet(T proceed()) )();

}
```

Note that `lang.Referenceable` is not a subclass of `lang.Object`.

The subtype `lang.Assignable` represents a reference to an assignable value, allowing pass by reference semantics for assignable values. Expressions of type `Assignable` support the binary operator `:=` (assign).

```
public abstract class Assignable<T>()
        extends Referenceable<T>() {

    doc "Assign the value, returning the new
         value after assignment. The binary
         assign operator |:=|."
    public T setIt(T t);

    public void intercept( void onSet(void proceed(T value), T value) )();

}
```

TODO: should `onSet()` return the new value of the attribute instead of being `void`?

6.5. Boolean values

The `lang.Boolean` class represents boolean values. Expressions of type `Boolean` support the binary `||, &&` and `=>` operators and the unary `!` operator, and inherit the binary `|, &, ^` and unary `~` operators from `Bits`.

```
public class Boolean()
        satisfies Case<Boolean>, Bits<Boolean> {
    case true, case false;

    doc "The binary or operator |x | y|"
    override public Boolean or(Boolean boolean) {
        if (this) {
            return true
        }
        else {
            return boolean
        }
    }

    doc "The binary and operator |x & y|"
    override public Boolean and(Boolean boolean) {
```

```
        if (this) {
            return boolean
        }
        else {
            return false
        }
    }

    doc "The binary xor operator |x ^ y|"
    override public Boolean xor(Boolean boolean) {
        if (this) {
            return boolean.complement
        }
        else {
            return boolean
        }
    }

    doc "The unary not operator |!x|"
    override public Boolean complement {
        if (this) {
            return false
        }
        else {
            return true
        }
    }

    override public List<Boolean> bits {
        return SingletonList(this)
    }
}
```

6.6. Cases and selectors

The interface `lang.Case` represents a type that may be used as a `case` in the `switch` construct.

```
public interface Case<in X> {

    doc "Determine if the given value matches
         this case, returning |true| iff the
         value matches."
    public Boolean test(X value);

}
```

TODO: the two extensions that follow don't actually work, since they overload the `test()` method of `Case` in a way that doesn't permit resolution according to the rules we have written down.

An extension allows `Case<X>` to function as `Case<X?>`.

```
public extension
class CaseOptional<X>(Case<X> c)
        satisfies Case<X?> {

    override Boolean test(X? x) {
        if (exists value) {
            return c.test(x)
        }
        else {
            return false;
        }
    }

}
```

An extension allows `Case<X>` to function as `Case<Y>` where `X` is assignable to `Y`.

```
public extension
class CaseSuper<X,Y>(Case<X> c)
        satisfies Case<Y>
        given X satisfies Y {

    override Boolean test(Y y) {
        if (is X y) {
            return c.test(y)
        }
        else {
```

```
            return false;
        }
    }

}
```

Classes with enumerated `case`s implicitly implement `lang.Selector`

```
public interface Selector {
    public String name;
    public small Integer ordinal;
}
```

The implementations of `equals()`, `hash` and `string` provided by the `Object` class handle subclasses that implement `Selector` as a special case.

This extension allows selectors to be used as cases in a `switch` statement:

```
public extension class SelectorCase<X>(X selector)
        given X extends Selector {
    override Boolean test(X x) {
        return x.ordinal==selector.ordinal
    }
}
```

Of course, the compiler is permitted to optimize away the call to this extension.

6.7. Optional and null values

The class `lang.Optional` represents a value that may be null.

```
abstract class Optional<out X>()
        extends Void() {

    doc "The unary postfix exists operator
         |x exists|."
    public Boolean valueExists;

    doc "The binary default operator
         |x ? y|."
    public T default<T>(T defaultValue)
            given T abstracts X;

}
```

Expressions of type `Object?` support the binary operators `?.` (nullsafe invoke) and `?` (default), the unary operator `exists`, the unary operator `$` (render) and the the binary operators `==` (equals) and `!=` (not equals).

Note that `lang.Optional` is not a subtype of `lang.Object`.

If an optional value is not null, it is represented by an instance of the subclass `lang.Something`.

```
public extension
class Something<out X>(X x)
        extends Optional<X>() {

    public extension X value = x;

    override Boolean valueExists {
        return true
    }

    override T default<T>(T defaultValue)
            given T abstracts X {
        return value;
    }

}
```

Non-optional values are transparently assignable to optional values, since `Something<X>` is an extension of `X` enabled in every compilation unit. Likewise, instances of `Something<X>` are transparently assignable to `X`.

If an optional value is null, it is represented by an instance of the subclass `lang.Nothing`.

```
public extension
class Nothing<out X>(Null null)
        extends Optional<X>() {

    override Boolean valueExists {
        return false
    }

    override T default<T>(T defaultValue)
            given T abstracts X {
        return defaultValue;
    }

}
```

The value `Null.null` is transparently assignable to optional values, since `Nothing<X>` is an extension of `Null` enabled in every compilation unit.

```
public class Null() {

    doc "Represents a null reference."
    case null;

}
```

The decorator `lang.NullCase` allows `null` to appear as a `case` expression in a `switch` statement.

```
public extension
class NullCase<X>(Null null) satisfies Case<Optional<X>> {

    override Boolean test(Optional<X> value) {
        return value is Nothing<X>;
    }

}
```

Note that `Optional` is not a reified type. The compiler erases all references to `Optional<X>` to `X` after performing type validation and before generating bytecode. The compiler also replaces references to `Null.null` with the Java `null`. Therefore, none of the extensions defined in this section are actually executed.

6.8. Usables

The interface `lang.Usable` represents an object with a lifecycle controlled by `try`.

```
public interface Usable {

    doc "Called before entry into a |try| block."
    public void begin();

    doc "Called before normal exit from a |try| block."
    public void end();

    doc "Called before exit from a |try| block when an
         exception occurs."
    public void end(Exception e);

}
```

6.9. Iterable objects and iterators

The interface `Iterable` represents an object which can produce a sequence of values, and which can be iterated using a `for` loop.

```
public interface Iterable<out X> {

    doc "A sequence of objects belonging
         to the container."
    public Iterator<X> iterator();

}
```

A Ceylon `Iterator` is a stateless object that produces a theoretically unbounded sequence of values.

```
public interface Iterator<out X> {
    public Reference<X>? head;
    public Iterator<X> tail;
}
```

An `Iterator` is used according to the following idiom:

```
mutable local iter := iterable.iterator();
while (exists local value = iter.head) {
    ...
    iter.=tail;
}
```

TODO: should we provide for mutation/removal during iteration:

```
public mutable interface OpenIterable<X>
        satisfies Iterable<X> {
    override OpenIterator<X> iterator();
}
```

```
public mutable interface OpenIterator<X>
        satisfies Iterator<X> {
    override Assignable<X>? head;
    public void remove();
}
```

TODO: this alternate solution abstracts efficient iteration of sequences and linked lists/trees, but suffers from the problem that the `indexedValue()` operation accepts tokens from iterations of other objects, and is therefore much less typesafe.

```
public interface Indexed<out X, I>
        given I satisfies Ordinal {

    public I firstIndex;
    public Referenceable<X>? indexedValue(I index);

}
```

For example, a `Sequence<X>` would be an `Indexed<X, Natural>` and a `LinkedList<X>` would be an `Indexed<X, Link<X>>`. The idiom would be:

```
mutable local i := indexed.firstIndex;
while (exists local value = indexed.indexedValue(i)) {
    ...
    ++i;
}
```

Mutation/removal during iteration would also be possible.

```
public mutable interface OpenIndexed<X, I>
        satisfies Indexed<X,I>
        given I satisfies Ordinal {
    override mutable Assignable<X>? indexedValue(I index);
    public void removeIndex(I index);
}
```

6.10. Containers and categories

The interface `lang.Container` represents the abstract notion of an object that may be empty. Expressions of type `Container?` support the unary postfix operator `nonempty`.

```
public interface Container {

    doc "The nonempty operator. Determine
         if the container is empty."
    public Boolean empty;

}
```

The interface `lang.Category` represents the abstract notion of an object that contains other objects.

```
public interface Category {

    doc "Determine if the given objects belong to the category.
         Return |true| iff all the given objects belong to the
         category."
    public Boolean contains(Object... objects);

}
```

There is a mutable subtype, representing a category to which objects may be added.

```
public mutable interface OpenCategory<in X>
        satisfies Category
        given X satisfies Object {

    doc "Add the given objects to the category. Return the number of
         objects which did not already belong to the category."
    public Natural add(X... objects);

}
```

6.11. Entries

The `Entry` class represents a pair of associated objects.

Entries may be constructed using the `->` operator.

```
public class Entry<out U, out V>(U key, V value) {

    doc "The key used to access the entry."
    public U key = key;

    doc "The value associated with the key."
    public V value = value;

    override public Boolean equals(Object that) {
        return equals(that, Entry.key, Entry.value)
    }

    override public Integer hash {
        return hash(Entry.key, Entry.value)
    }

}
```

6.12. Correspondences

The interface `lang.Correspondence` represents the abstract notion of an object that maps value of one type to values of some other type. It supports the binary operator `[key]` (lookup).

```
public interface Correspondence<in U, out V>
        given U satisfies Object {

    doc "Binary lookup operator x[key]. Returns the value defined
         for the given key, or |null| if there is no value defined
         for the given key."
    public Referenceable<V?> value(U key);

}
```

A decorator allows retrieval of lists and sets of values.

```
public extension class Correspondences<in U, out V>(Correspondence<U, V> correspondence)
        given U satisfies Object {

    doc "Binary sequenced lookup operator |x[keys]|. Return a list
         of values defined for the given keys, in order."
    throws (UndefinedKeyException
            -> "if no value is defined for one of the given
                keys")
    public V[] values(U... keys) {
        return from (U key in keys) select (correspondence[key])
```

```
    }

    doc "Binary iterated lookup operator |x[keys]|. Return an iterator
         of the values defined for the given keys."
    throws (UndefinedKeyException
            -> "if no value is defined for one of the given
                keys")
    public Iterable<V> values(Iterable<U> keys) {
        return from (U key in keys) select (correspondence[key])
    }

    doc "Determine if there are values defined for the given keys.
         Return |true| iff there are values defined for all the
         given keys."
    public Boolean defines(U... keys) {
        return forAll (U key in keys) every (correspondence[key] exists)
    }

}
```

There is a mutable subtype, representing a correspondence for which new mappings may be defined, and existing mappings modified. It provides for the use of element expressions in assignments.

```
public mutable interface OpenCorrespondence<in U, V>
        satisfies Correspondence<U, V>
        given U satisfies Object {

    override public Assignable<V?> value(U key);

}
```

A decorator allows addition of multiple `Entry`s.

```
public extension class OpenCorrespondences<in U, V>(OpenCorrespondence<U, V> correspondence)
        given U satisfies Object {

    doc "Add the given entries, overriding any definitions that
         already exist."
    throws (UndefinedKeyException
            -> "if a value can not be defined for one of the
                given keys")
    public void define(Entry<U, V>... definitions) {
        for (U key->V value in definitions) {
            correspondence[key] := value;
        }
    }

    doc "Assign a value to the given key. Return the previous value
         for the key, or |null| if there was no value defined."
    throws (UndefinedKeyException
            -> "if a value can not be defined for the given key")
    public V? define(U key, V value) {
        Assignable<V?> definition = correspondence[key];
        V? result = definition;
        definition := value;
        return result;
    }

}
```

6.13. Sequences

A `lang.Sequence` is a correspondence from a bounded progression of natural numbers. Sequences support the binary operators `[].` (spread), `+` (join) and `[i...]` (upper range) and the ternary operator `[i..j]` (subrange) in addition to operators inherited from `Correspondence`:

```
public interface Sequence<out X>
        satisfies Correspondence<Natural, X> {

    doc "The index of the last element of the sequence,
         or |null| if the sequence has no elements."
    public Natural? lastIndex;

}
```

Any `Sequence` is `Iterable` and is a `Container`, according to the following decorator:

```
public extension
SequenceIterable<X>(X[] sequence)
        satisfies Iterable<X>, Container {
    override Iterator<X> iterator() {
        class SequenceIterator(Natural from)
                satisfies Iterator<X> {
            override X? head {
                return sequence[from]
            }
            override Iterable<X> tail {
                return SequenceIterator(from+1)
            }
        }
        if (is Iterable<X> sequence) {
            return sequence.iterator()
        else {
            return SequenceIterator(0)
        }
    }
    override Boolean empty {
        return !(sequence.lastIndex exists)
    }
}
```

The following extension provides access to the indices of the sequence in a `for` loop.

```
public extension
class SequenceEntryIterable<X>(X[] sequence)
        satisfies Iterable<Entry<Natural,X>> {
    override Iterator<Entry<Natural,X>> iterator() {
        class EntryIterator(Natural from)
                satisfies Iterator<Entry<Natural,X>> {
            override Entry<Natural,X>? head {
                if (exists X x = sequence[from]) {
                    return from->x
                }
                else {
                    return null;
                }
            }
            override Iterable<Entry<Natural,X>> tail {
                return EntryIterator(from+1)
            }
        }
        return EntryIterator(0)
    }
}
```

TODO: or should we just directly make `X[]` assignable to `Entry<Natural,X>[]`?

The compiler is permitted to optimize away the call to these extensions in a loop such as:

```
for (local elem in sequence) {
    ...
}
```

or even:

```
for (Natural n -> local elem in sequence) {
    ...
}
```

which are both equivalent to:

```
if (exists Natural last=sequence.lastIndex) {
    mutable Natural n := 0;
    while (n<=last) {
        local elem = sequence[n];
        ...
        ++n;
    }
}
```

TODO: for some kinds of sequences, especially linked lists, access by index is very inefficient. How can we make iteration of these kinds of sequences efficient?

The value `None.none` is transparently assignable to `Sequence<X>`.

```
public class None() {

    doc "Represents an empty sequence. The value of
         the empty sequence enumeration |{}|."
    case none;

    public extension
    class EmptySequence<out X>() satisfies X[] {
        override Natural? lastIndex = null;
        override Referenceable<X?> value(Natural index) {
            X? nullValue = null;
            return nullValue
        }
    }

}
```

Toplevel methods define the join and range operators, for sequence types:

```
doc "The binary join operator |x + y|. The returned
     sequence does not reflect changes to the original
     sequences."
public T[] join<T>(T[]... sequences)  {

    class JoinedSequence()
            satisfies T[] {
        override Natural? lastIndex {
            mutable Natural? result := null;
            for (T[] s in sequences) {
                if (exists Natural last = s.lastIndex) {
                    if (exists result) {
                        result += last;
                    }
                    else {
                        result := last;
                    }
                }
            }
            return result
        }
        override Referenceable<T?> value(Natural index) {
            T? value {
                mutable Natural i := index;
                for (T[] s in sequences) {
                    if (exists Natural last = s.lastIndex) {
                        if (i<=last) {
                            return s[i]
                        }
                        else {
                            i-=last;
                        }
                    }
                }
                return null

            }
            return value
        }
    }

    return copy(JoinedSequence()) //take a shallow copy

}
```

```
doc "The ternary range operator |x[from..to]|, along
     with the binary upper range |x[from...]| operator.
     The returned sequence does not reflect changes
     to the original sequence."
public T[] range<T>(T[] sequence, Natural from, Natural? to=sequence.lastIndex) {

    class RangeSequence()
            satisfies T[] {
        override Natural? lastIndex {
            if (exists Natural last = sequence.lastIndex) {
                if (exists to) {
                    if (to<last) {
                        return to-from
                    }
                    else {
                        return last-from
                    }
                }
            }
```

```
            else {
                return null
            }
        }
        override Referenceable<T?> value(Natural index) {
            T? value {
                if (exists to) {
                    if (index>to) {
                        return null
                    }
                }
                return sequence[index+from]
            }
            return value
        }
     }

    return copy(RangeSequence()) //take a shallow copy

}
```

A helper class decorates `Sequence` with some convenience attributes:

```
public extension class Sequences<out X>(X[] sequence) {

    doc "The first element of the sequence, or
         |null| if the sequence has no elements."
    public X? first {
        return sequence[0]
    }

    doc "The rest of the sequence, after removing
         the first element."
    public X[] rest {
        return sequence[1...]
    }

    doc "The last element of the sequence, or
         |null| if the sequence has no elements.."
    public X? last {
        if (exists Natural index = sequence.lastIndex) {
            return sequence[index]
        }
        else {
            return null
        }
    }

}
```

The `zip()` function combines `Sequences` into a single `Sequence`.

```
public T[] zip<T,X,Y>(X[] x, Y[] y, T producing(X x, Y y)) {
    return from(Natural i in 0..min(x.lastIndex,y.lastIndex))
                select (f(x[i],y[i]))
}
```

```
public Entry<X,Y>[] zip<X,Y>(X[] x, Y[] y) {
    return zip(x,y,Entry)
}
```

```
public T[] zip<T,X>(X[] lists..., T producing(X x...)) {
    Natural len = min(from (X[] list in lists) select (list.lastIndex));
    return from (Natural i in 0..len)
                select (f(from (X[] list in lists) select (list[i]));
}
```

```
public T[] zip<T,X>(X[] lists...) {
    return zip(lists,ArrayList)
}
```

There is a mutable subtype which allows assignment to an index.

```
public mutable interface OpenSequence<X>
        satisfies X[], OpenCorrespondence<Natural,X> {}
```

6.14. Collections

The interface `lang.Collection` is the root of the Ceylon collections framework.

```
public interface Collection<out X>
        satisfies Iterable<X>, Container, Category
        given X satisfies Object {

    doc "The number of elements or entries belonging to the
         collection."
    public Natural size;

    doc "Determine the number of times the given element
         appears in the collection."
    public Natural count(Object element);

    doc "Determine the number of elements or entries for
         which the given condition evaluates to |true|."
    public Natural count(Boolean where(X element));

    doc "Determine if the given condition evaluates to |true|
         for at least one element or entry."
    public Boolean contains(Boolean where(X element));

    doc "The elements of the collection, as a |Set|."
    public Set<X> elements;

    doc "The elements of the collection for which the given
         condition evaluates to |true|, as a |Set|."
    public Set<X> elements(Boolean where(X element));

    doc "The elements of the collection, sorted using the given
         comparison."
    public List<X> sortedElements(Comparison by(X x, X y));

    doc "An extension of the collection, with the given
         elements. The returned collection reflects changes
         made to the first collection."
    public Collection<T> with<T>(T... elements) given T abstracts X;

    doc "A mutable copy of the collection."
    public OpenCollection<T> copy<T>() given T abstracts X;

}
```

A decorator provides the ability to sort collections of `Comparable` values in natural order.

```
public extension class CollectionsOfComparable<out X>(Collection<X> collection)
        given X satisfies Comparable<X> {

    doc "The elements of the collection, sorted in natural order."
    public List<X> sortedElements() {
        return collection.sortedElements() by (X x, X y) (x<=>y)
    }

}
```

Mutable collections implement `lang.OpenCollection`:

```
public mutable interface OpenCollection<X>
        satisfies OpenCategory<X>, Collection<X>
        given X satisfies Object {

    doc "Remove all elements or entries of the collection,
         resulting in an empty collection."
    public Boolean clear();

    doc "Remove the given elements from the collection.
         Return the number of elements which belonged
         to the collection."
    public Natural remove(X... elements);

    doc "Remove all elements from the collection for which
         the given condition evaluates to |true|. Return
         the number of elements which were removed."
    public Natural remove(Boolean where(X element));

}
```

6.14.1. Sets

Sets implement the following interface:

```
public interface Set<out X>
        satisfies Collection<X>, Correspondence<Object, Boolean>
        given X satisfies Object {

    doc "Determine if the set is a superset of the given set.
         Return |true| if it is a superset."
    public Boolean superset(Set<Object> set);

    doc "Determine if the set is a subset of the given set.
         Return |true| if it is a subset."
    public Boolean subset(Set<Object> set);

    public override Set<T> with<T>(T... elements) given T abstracts X;
    public override OpenSet<T> copy<T>() given T abstracts X;

}
```

There is a mutable subtype:

```
public mutable interface OpenSet<X>
        satisfies Set<X>, OpenCollection<X>, OpenCorrespondence<Object, Boolean>
        given X satisfies Object {}
```

6.14.2. Lists

Lists implement the following interface, and support the operators inherited from `Collection` and `Sequence`:

```
public interface List<out X>
        satisfies Collection<X>, X[]
        given X satisfies Object {

    doc "The index of the first element of the list
         which satisfies the condition, or |null| if
         no element satsfies the condition."
    public Natural? firstIndex(Boolean where(X element));

    doc "The index of the last element of the list
         which satisfies the condition, or |null| if
         no element satsfies the condition."
    public Natural? lastIndex(Boolean where(X element));

    doc "The elements of the list for which the given condition
         evaluates to |true|, as a |List| with the original
         order."
    public List<X> elementList(Boolean where(X element));

    doc "A sublist beginning with the element at the first given
         index up to and including the element at the second given
         index. The size of the returned sublist is one more than
         the difference between the two indexes. The returned list
         does reflect changes to the original list."
    public List<X> sublist(Natural from, Natural to=lastIndex);

    doc "A sublist of the given length beginning with the
         first element of the list. The returned list does
         reflect changes to the original list."
    public List<X> leading(Natural length=1);

    doc "A sublist of the given length ending with the
         last element of the list. The returned list does
         reflect changes to the original list."
    public List<X> trailing(Natural length=1);

    doc "Split the list into sublists, each beginning at an
         element for which the predicate returns true."
    public Iterable<List<X>> sublists(Boolean split(X element));

    doc "Split the list into sublists, each beginning at an
         element for which the predicate returns true. The
         predicate may examine elements to the left and
         right of the element under consideration."
    public Iterable<List<X>> sublists(
            Boolean split(
                      doc "The current element, which
                           will be the first element
                           of the sublist if |split()|
```

```
                        returns |true|."
                    X element,
                    doc "The elements to the left,
                         beginning with the immediately
                         adjacent element."
                    X[] left,
                    doc "The elements to the right,
                         beginning with the immediately
                         adjacent element."
                    X[] right
                )
            );

    doc "An extension of the list with the given elements
         at the end of the list. The returned list does
         reflect changes to the original list."
    public override List<T> with<T>(T... elements) given T abstracts X;

    doc "An extension of the list with the given elements
         at the start of the list. The returned list does
         reflect changes to the original list."
    public List<T> withInitial<T>(T... elements) given T abstracts X;

    doc "The list in reverse order. The returned list does
         reflect changes to the original list."
    public List<X> reversed;

    doc "The unsorted elements of the list. The returned
         bag does reflect changes to the original list."
    public Bag<X> unsorted;

    doc "A map from list index to element. The returned
         map does reflect changes to the original list."
    public Map<Natural,X> map;

    doc "Produce a new list by applying an operation to
         every element of the list."
    public List<Y> transform<Y>(Y select(X element));

    public override OpenList<T> copy<T>() given T abstracts X;

}
```

Any `Sequence` may be transparently converted to a `List`:

```
public extension List<X> sequenceToList(X[] sequence) {
    if (is List<X> sequence) {
        return sequence
    }
    else {
        return SequenceList(sequence)
    }
}
```

There is a mutable subtype:

```
public mutable interface OpenList<X>
        satisfies List<X>, OpenCollection<X>, OpenSequence<X>
        given X satisfies Object {

    doc "Remove the element at the given index, decrementing
         the index of every element with an index greater
         than the given index by one. Return the removed
         element."
    throws (UndefinedKeyException
            -> "if the index is not in the list")
    public X removeIndex(Natural at);

    doc "Add the given elements at the end of the list."
    public void append(X... elements);

    doc "Add the given elements at the start of the list,
         incrementing the index of every existing element
         by the number of given elements."
    public void prepend(X... elements);

    doc "Insert the given elements beginning at the given
         index, incrementing the index of every existing
         element with that index or greater by the number
         of given elements."
    throws (UndefinedKeyException
            -> "if the index is not in the list")
    public void insert(Natural at, X... elements);
```

```
    doc "Remove elements beginning with the first given index
         up to and including the second given index,
         decrementing the indexes of all elements after
         with the second given index by one more than the
         difference between the two indexes."
    throws (UndefinedKeyException
            -> "if the index is not in the list")
    public void delete(Natural from, Natural to=lastIndex);

    doc "Remove and return the first element, decrementing
         the index of every other element by one."
    throws (EmptyException
            -> "if the list is empty")
    public X removeFirst();

    doc "Remove and return the last element."
    throws (EmptyException
            -> "if the list is empty")
    public X removeLast();

    doc "Reverse the order of the list."
    public void reverse();

    doc "Reorder the elements of the list, according to the
         given comparison."
    public void resort(Comparison by(X x, X y));

    override public OpenList<X> leading(Natural length);
    override public OpenList<X> trailing(Natural length);
    override public OpenList<X> sublist(Natural from, Natural to);
    override public OpenList<X> reversed;
    override public OpenMap<Natural,X> map;

}
```

A decorator provides the ability to resort lists of `Comparable` values in natural order.

```
public extension class OpenListsOfComparable<out X>(OpenList<X> list)
        given X satisfies Comparable<X> {

    doc "Reorder the elements of the list, according to the
         natural order."
    public void resort() {
        list.resort() by (X x, X y) (x<=>y);
    }

}
```

6.14.3. Maps

Maps implement the following interface:

TODO: is it OK that maps are not contravariant in the key type?

```
public interface Map<U, out V>
        satisfies Collection<Entry<U,V>>, Correspondence<U, V>
        given U satisfies Object {

    doc "The keys of the map, as a |Set|."
    public Set<U> keys;

    doc "The values of the map, as a |Bag|."
    public Bag<V> values;

    doc "A |Map| of each value belonging to the map, to the
         |Set| of all keys at which that value occurs."
    public Map<V, Set<U>> inverse;

    doc "Produce a new map by applying an operation to every
         element of the map."
    public Map<U, W> transform<W>(W? select(U key -> V value));

    doc "The entries of the map for which the given condition
         evaluates to |true|, as a |Map|."
    public Map<U, V> entries(Boolean where(U key -> V value));

    public override Map<U, T> with<T>(Entry<U, T>... entries) given T abstracts V;
    public override OpenMap<U,T> copy<T>() given T abstracts V;
```

```
}
```

There is a mutable subtype:

```
public mutable interface OpenMap<U,V>
        satisfies Map<U,V>, OpenCollection<Entry<U,V>>, OpenCorrespondence<U, V>
        given U satisfies Object {

    doc "Remove the entry for the given key, returning the
         value of the removed entry."
    throws (UndefinedKeyException
            -> "if no value is defined for the given key")
    public V remove(U key);

    doc "Remove all entries from the map which have keys for
         which the given condition evaluates to |true|. Return
         entries which were removed."
    public Map<U,V> remove(Boolean where(U key));

    override public OpenSet<U> keys;
    override public OpenBag<V> values;
    override public OpenMap<V, Set<U>> inverse;

}
```

6.14.4. Bags

Bags implement the following interface:

```
public interface Bag<out X>
        satisfies Collection<X>, Correspondence<Object, Natural>
        given X satisfies Object {

    doc "A map from element to the number of occurrences of
         the element. The returned map reflects changes to
         the original bag."
    public Map<X,Natural> map;

    public override Bag<T> with<T>(T... elements) given T abstracts X;
    public override OpenBag<T> copy<T>() given T abstracts X;

}
```

There is a mutable subtype:

```
public mutable interface OpenBag<X>
        satisfies Bag<X>, OpenCollection<X>, OpenCorrespondence<Object, Natural> {

    override public OpenMap<X,Natural> map;

}
```

6.15. Ordered values

The `lang.Comparable` interface represents totally ordered types, and supports the binary operators `>`, `<`, `<=`, `>=` and `<=>` (compare).

```
public interface Comparable<in T>
        given T satisfies Object {

    doc "The binary compare operator |<=>|. Compares this
         object with the given object."
    public Comparison compare(T other);

}
```

```
public class Comparison() {

    doc "The receiving object is larger than
         the given object."
    case larger,

    doc "The receiving object is smaller than
         the given object."
```

```
    case smaller,

    doc "The receiving object is exactly equal
         to the given object."
    case equal;

    public Boolean larger { return this===larger }
    public Boolean smaller { return this===smaller }
    public Boolean equal { return this===equal }
    public Boolean unequal { return !this===equal }
    public Boolean largeAs { return !this===smaller }
    public Boolean smallAs { return !this===larger }

}
```

TODO: should we support partial orders? Give `Comparison` an extra `uncomparable` value, or let `compare()` return `null`?

TODO: if so, why not just move `compare()` up to `Object` to simplify things?

The toplevel methods `min()` and `max()` return the minimum and maximum values of a list of `Comparable`s.

```
public X min<X>(X x, X... xs)
        given X satisfies Comparable<X> {
    mutable X min := x;
    for (X y in xs) {
        if (y<min) {
            min:=y;
        }
    }
    return min
}
```

```
public X max<X>(X x, X... xs)
        given X satisfies Comparable<X> {
    mutable X max := x;
    for (X y in xs) {
        if (y>max) {
            max:=y;
        }
    }
    return max
}
```

The `lang.Ordinal` interface represents objects in a sequence, and supports the binary operator `..` (range). In addition, variables support the postfix unary operators `++` (increment) and `--` (decrement) and prefix unary operators `++` (successor) and `--` (predecessor).

```
public interface Ordinal {

    doc "The unary |++| operator. The successor of this instance."
    throws (OutOfRangeException
            -> "if this is the maximum value")
    public subtype successor;

    doc "The unary |--| operator. The predecessor of this instance."
    throws (OutOfRangeException
            -> "if this is the minimum value")
    public subtype predecessor;

}
```

6.16. Ranges

Ranges implement `Sequence`, therefore they support the join, subrange, contains and lookup operators, among others. Ranges may be constructed using the `..` operator:

```
public class Range<X>(X first, X last)
        satisfies Category, X[], Case<X>
        given X satisfies Ordinal, Comparable<X> {

    doc "The first value in the range."
    public X first = first;

    doc "The last value in the range."
    public X last = last;
```

```
    ...

    doc "Return a |Sequence| of values in the range,
         beginning at the first value, and
         incrementing by a constant step size,
         until a value outside the range is
         reached."
    public X[] by(Natural stepSize) { return ... }

}
```

The compiler is permitted to optimize away creation of a `Range` in code such as the following:

```
i in min..max
```

which is equivalent to:

```
(i>=min && i<=max)
```

and:

```
for (local i in min..max) {
    ...
}
```

which is equivalent to:

```
mutable local i := min;
while (i<=max) {
    ...
    i.=successor;
}
```

6.17. Characters and strings

UTF-32 Unicode Characters are represented by the following class:

```
public class Character(small Natural utf32)
        satisfies Ordinal, Comparable<Character>, Case<Character> {
    ...

    doc "The UTF-16 encoding"
    public String utf16;

    doc "The UTF-8 encoding"
    public String utf8;

    public extension class StringToCharacter(String string) {

        doc "Parse the string representation of a |Character| in UTF-16"
        public Character parseUtf16Character() { return ... }

        doc "Parse the string representation of a |Character| in UTF-8"
        public Character parseUtf8Character() { return ... }

    }

}
```

`String` implements `Sequence`, and therefore supports the join, subrange, contains and lookup operators, among others. Any `Character[]` may be transparently converted to `String`.

```
public extension
class String(Character[] characters)
        satisfies Character[], Comparable<String>, Case<String> {

    ...

    doc "Split the string into tokens, using the given
         separator characters."
    public Iterable<String> tokens(Iterable<Character> separators=" ,;\n\l\r\t") { return ... }

    doc "Split the string into lines of text."
    public Iterable<String> lines() { return tokens("\n\l\r") }
```

```
    doc "The string, with all characters in lowercase."
    public String lowercase { return ... }

    doc "The string, with all characters in uppercase."
    public String uppercase { return ... }

    doc "Remove the given characters from the beginning
         and end of the string.
    public String strip(Character[] whitespace = " \n\l\r\t") { return ... }

    doc "Collapse substrings of the given characters into
         single space characters."
    public String normalize(Character[] whitespace = " \n\l\r\t") { return ... }

    doc "Join the given strings, using this string as
         a separator."
    public String join(String... strings) { return ... }

}
```

What encoding is the default for Ceylon strings? Are there performance advantages to going with UTF-16 like Java? Can we easily abstract this stuff? Does the JVM do all kinds of optimizations for `java.lang.String`?

A string template is represented by an instance of `StringTemplate`.

```
public class StringTemplate(...) {

    doc "Evaluate all interpolated expressions, producing
         a constant character string with no interpolated
         expressions."
    public extension String interpolate() { return ... }

}
```

6.18. Regular expressions

```
public extension class Regex(Quoted expression)
        satisfies Case<String> {

    doc "Return the substrings of the given string which
         match the parenthesized groups of the regex,
         ordered by the position of the opening parenthesis
         of the group."
    public Match? matchList(String string)() { return ... }

    doc "Determine if the given string matches the regex."
    public Boolean matches(String string) { return ... }

    ...

}
```

TODO: I assume these are just Java (Perl 5-style) regular expressions. Is there some other better syntax around?

6.19. Bit strings

The interface `Bits` represents a fixed length string of boolean values, and supports the binary `|, &, ^` and unary `~` operators.

```
public interface Bits<T> {

    doc "Bitwise or operator |x | y|"
    public T or(T bits);

    doc "Bitwise and operator |x & y|"
    public T and(T bits);

    doc "Bitwise xor operator |x ^ y|"
    public T xor(T bits);

    doc "Bitwise complement operator |~x|"
    public T complement;

    doc "A list of the bits."
```

```
    public List<Boolean> bits;

}
```

6.20. Numbers

The `lang.Number` interface is the abstract supertype of all classes which represent numeric values.

```
public interface Number {

    doc "Determine if the number represents
         an integer value"
    public Boolean integral;

    doc "Determine if the number is positive"
    public Boolean positive;

    doc "Determine if the number is negative"
    public Boolean negative;

    doc "Determine if the number is zero"
    public Boolean zero;

    doc "Determine if the number is one"
    public Boolean unit;

    doc "The number, represented as a |Decimal|"
    public Decimal decimal;

    doc "The number, represented as a |Float|"
    throws (FloatOverflowException
            -> "if the number is too large to be
                represented as a |Float|")
    public Float float;

    doc "The number, represented as an |Whole|,
         after truncation of any fractional
         part"
    public Whole whole;

    doc "The number, represented as an |Integer|,
         after truncation of any fractional
         part"
    throws (IntegerOverflowException
            -> "if the number is too large to be
                represented as an |Integer|")
    public Integer integer;

    doc "The number, represented as a |Natural|,
         after truncation of any fractional
         part"
    throws (NegativeNumberException
            -> "if the number is negative")
    public Natural natural;

    doc "The magnitude of the number"
    public subtype magnitude;

    doc "1 if the number is positive, -1 if it
         is negative, or 0 if it is zero."
    public subtype sign;

    doc "The fractional part of the number,
         after truncation of the integral
         part"
    public subtype fractionalPart;

    doc "The integral value of the number
         after truncation of the fractional
         part"
    public subtype wholePart;

}
```

The subtype `lang.Numeric` supports the binary operators `+`,`-`, `*`, `/`, `**`. In addition, `mutable` values of type `Numeric` support the compound assignment operators `+=`, `-=`, `/=`, `*=`.

```
public interface Numeric<N>
        satisfies Number, Comparable<N>
        given N satisfies Number {
```

```
    doc "The binary |+| operator"
    public N plus(N number);

    doc "The binary |-| operator"
    public N minus(N number);

    doc "The binary |*| operator"
    public N times(N number);

    doc "The binary |/| operator"
    public N divided(N number);

    doc "The binary |**| operator"
    public N power(N number);

}
```

TODO: should `plus()` and `times()` accept varargs, to minimize method calls?

TODO: I suppose `Numeric` should not extend `Comparable`, since complex numbers are not comparable.

The subtype `lang.Integral` supports the binary operator `%`, and inherits the unary operators `++` and `--` from `Ordinal`.

```
public interface Integral<N>
        satisfies Numeric<N>, Ordinal
        given N satisfies Number {

    doc "The binary |%| operator"
    public N remainder(N number);

}
```

The type `lang.Invertable` supports the unary prefix `-` operator.

```
public interface Invertable<I>
    given I satisfies Number {

    doc "The unary |-| operator"
    public I inverse;

}
```

Five numeric types are built in:

`lang.Natural` represents 63 bit unsigned integers (including zero).

```
public class Natural(Natural natural)
        satisfies Integral<Natural>, Invertable<Integer>, Case<Integer>, Bits<Natural> {
    ...

    doc "Implicit type promotion to |Integer|"
    override public extension Integer integer { return ... }

    doc "Implicit type promotion to |Whole|"
    override public extension Whole whole { return ... }

    doc "Implicit type promotion to |Float|"
    override public extension Float float { return ... }

    doc "Implicit type promotion to |Decimal|"
    override public extension Decimal decimal { return ... }

    doc "Shift bits left by the given number of places"
    public Natural leftShift(Natural digits) { return ... }

    doc "Shift bits right by the given number of places"
    public Natural rightShift(Natural digits) { return ... }

    public extension class StringToNatural(String string) {

        doc "Parse the string representation of a |Natural| in the given radix"
        public Natural parseNatural(small Natural radix=10) { return ... }

    }

}
```

TODO: the `Numeric` type is much more complicated because of `Natural`. As an alternative approach, we could replace `Natural` with a `Binary` type, and have `Integer` literals instead of `Natural` literals. I don't love this, because natural numbers are especially common in real applications.

`lang.Integer` represents 64 bit signed integers.

```
public class Integer(Boolean sign, Natural natural)
        satisfies Integral<Integer>, Invertable<Integer>, Case<Integer> {
    ...

    doc "Implicit type promotion to |Whole|"
    override public extension Whole whole { return ... }

    doc "Implicit type promotion to |Float|"
    override public extension Float float { return ... }

    doc "Implicit type promotion to |Decimal|"
    override public extension Decimal decimal { return ... }

    public extension class StringToInteger(String string) {

        doc "Parse the string representation of an |Integer| in the given radix"
        public Integer parseInteger(small Natural radix=10) { return ... }

    }

}
```

`lang.Whole` represents arbitrary-precision signed integers.

```
public class Whole(Boolean sign, small Natural... digits)
        satisfies Integral<Whole>, Invertable<Whole> {
    ...

    public small Natural precision = ...;

    doc "Implicit type promotion to |Decimal|"
    override public extension Decimal decimal { return ... }

    public extension class StringToWhole(String string) {

        doc "Parse the string representation of a |Whole| in the given radix"
        public Whole parseWhole(small Natural radix=10) { return ... }

    }

}
```

`lang.Float` represents 64 bit floating point values.

```
public class Float(Float float)
        satisfies Numeric<Float>, Invertable<Float> {
    ...

    doc "Implicit type promotion to |Decimal|"
    override public extension Decimal decimal { return ... }

    public extension class StringToFloat(String string) {

        doc "Parse the string representation of a |Float| in the given radix"
        public Float parseFloat(small Natural radix=10) { return ... }

    }

}
```

`lang.Decimal` represents arbitrary-precision and arbitrary-scale decimals.

```
public class Decimal(Whole value, small Integer scale)
        satisfies Numeric<Decimal>, Invertable<Decimal> {
    ...

    public small Natural precision = ...;
    public small Integer scale = ...;

    public extension class StringToDecimal(String string) {

        doc "Parse the string representation of a |Decimal| in the given radix"
        public Decimal parseDecimal(small Natural radix=10) { return ... }
```

```
    }

}
```

6.21. Metamodel

The metamodel (reflection API) reifies the schema of a type, making it available to the program at runtime.

A `Function` represents a toplevel method.

```
public interface Function<R, P...>
        satisfies Callable<R, P...>, Annotated {

    public String name;

    public Boolean extension;

    public Visiblity visibility;

    public Type<R> returnType;
    public Parameter<Object>[] parameters;

}
```

An instance of `Type` represent a type: an interface, class or alias, together with type arguments.

```
public interface Type<out X>
        satisfies Annotated {

    public String name;

    public Boolean mutable;

    public Visiblity visibility;

    doc "Return all the attributes of the given
         type."
    public Set<Attribute<X,T>> attributes<T>(Type<T> type = Object);

    doc "Return all the methods of the given
         callable type."
    public Set<Method<X,R,P...>> methods<R,P...>(Type<Callable<R,P...>> type);

    doc "Return all the member classes of the
         given callable type."
    public Set<MemberClass<X,Y,P...>> classes<Y,P...>(Type<Callable<Y,P...>> type);

    ...

}
```

An instance of `Interface` represents an interface.

```
public interface Interface<X>
        satisfies Type<X> {}
```

An instance of `Class` represents a class.

```
public interface Class<X, P...>
        satisfies Type<X>, Callable<X,P...> {

    public Boolean abstract;
    public Boolean extension;

    public Parameter<Object>[] parameters;

}
```

An instance of `Member` represents a method or attribute.

```
public interface Member<in X, out T>
        satisfies Annotated, Callable<T,X> {

    public String name;
```

```
    public Boolean abstract;
    public Boolean default;
    public Boolean default;
    public Boolean override;
    public Boolean extension;

    public Visiblity visibility;

    public Type<T> memberType;

}
```

A `Method` represents a method declaration.

```
public interface Method<in X, R, P...>
        satisfies Member<Function<R, P...>> {

    public Type<R> returnType;
    public Parameter<Object>[] parameters;

    public void intercept<S>( R onInvoke(S instance, R proceed(P... args), P... args) )()
                    given S abstracts X;

}
```

An `Attribute` represents an attribute declaration.

```
public interface Attribute<in X, T>
        satisfies Member<X, Referenceable<T>> {

    public Boolean mutable;

    public void intercept<S>( T onGet(S instance, T proceed()) )()
                    given S abstracts X;

}
```

```
public interface MutableAttribute<in X, T>
        extends Attribute<X, T>
        satisfies Member<X, Assignable<T>> {  //this is OK since Member is covariant in T

    public void intercept<S>( void onSet(S instance, void proceed(T value), T value) )()
                    given S abstracts X;

}
```

An instance of `MemberClass` represents a member class.

```
public interface MemberClass<X, Y, P...>
        satisfies Type<Y>, Member<X, Class<Y,P...>> {

    public Parameter<Object>[] parameters;

    public void intercept<S>( Y onCreate(S instance, Y proceed(P... args), P... args) )()
                    given S abstracts X;

}
```

A `Parameter` represents a formal parameter of a method or class.

```
public interface Parameter<out T>
        satisfies Annotated {
    public String name;
    public Type<T> type;
}
```

An `Annotated` program element may be asked for a list of its annotation values.

```
public interface Annotated {
    doc "Return all the annotation values that are
         assignable to the given type."
    public T[] annotations<T>(Type<T> type = Object)
            given T satisfies Object;
}
```

6.22. Instants, intervals and durations

TODO: this stuff is just for illustration, the real date/time API will be much more complex and fully internationalized.

```
public class Instant() {
    ...
}
```

```
public class Time(Natural hours, Natural minutes,
        Natural? seconds=null, Natural? milliseconds=null,
        Timezone? timezone=null)
        extends Instant() {
    public Natural hours = hours;
    public Natural minutes = minutes;
    public Natural? seconds = seconds;
    public Natural? milliseconds = milliseconds;
    public Timezone? timezone = timezone;
    ...
}
```

```
public class Date(Integer year, Natural month, Natural day)
        extends Instant() {
    public Integer year = year;
    public Natural month = month;
    public Natural day = day;
    ...
}
```

```
public class Datetime(Time time, Date date)
        extends Instant() {
    public Time time = time;
    public Date date = date;
    ...
}
```

```
public class Interval<X>(X start, X end)
        given X satisfies Instant {
    public X start = start;
    public X end = end;
    public Duration<X> duration { return ... }
    ...
}
```

```
public class Duration<X>(Map<Granularity<X>, Natural> magnitude)
        given X satisfies Instant {

    public Map<Granularity<X>, Natural> magnitude = magnitude;

    public X before(X instant) { ... }
    public X after(X instant) { ... }

    public Datetime before(Datetime instant) { ... }
    public Datetime after(Datetime instant) { ... }

    public Duration<X> add(Duration<X> duration) { ... }
    public Duration<X> subtract(Duration<X> duration) { ... }

    ...
}
```

```
public interface Granularity<X>
        given X satisfies Instant {}
```

```
public class DateGranularity()
        satisfies Granularity<Date> {
    case year,
    case month,
    case week,
    case day;
}
```

```
public class TimeGranularity()
        satisfies Granularity<Time> {
    case hour,
    case minute,
    case second,
```

```
    case millisecond;
}
```

6.23. Control expressions

The `lang.assertion` package defines support for assertions.

```
doc "Assert that the block evaluates to true. The block
     is executed only when assertions are enabled. If
     the block evaluates to false, throw an
     |AssertionException| with the given message."
public void assert(StringTemplate message, Boolean that()) {
    if ( assertionsEnabled() && !that() ) {
        throw AssertionException(message)
    }
}
```

The `lang.conditional` package defines support for conditional expressions.

```
doc "If the condition is true, evaluate first block,
     and return the result. Otherwise, return |null|."
public Y? ifTrue<Y>(Boolean condition,
                    Y then()) {
    if (condition) {
        return then()
    }
    else {
        return null
    }
}
```

```
doc "If the condition is true, evaluate first block,
     otherwise, evaluate second block. Return result
     of evaluation."
public Y ifTrue<Y>(Boolean condition,
                   Y then(),
                   Y otherwise()) {
    if (condition) {
        return then()
    }
    else {
        return otherwise()
    }
}
```

```
doc "If the value is non-null, evaluate first block,
     and return the result. Otherwise, return |null|."
public Y? ifExists<X,Y>(specified X? value,
                        Y then(coordinated X x)) {
    if (exists value) {
        return then(value)
    }
    else {
        return null
    }
}
```

```
doc "If the value is non-null, evaluate first block,
     otherwise, evaluate second block. Return result
     of evaluation."
public Y ifExists<X,Y>(specified X? value,
                       Y then(coordinated X x),
                       Y otherwise()) {
    if (exists value) {
        return then(value)
    }
    else {
        return otherwise()
    }
}
```

The `lang.exceptional` package defines support for exceptional expressions.

```
doc "Attempt to evaluate the first block. If an
     exception occurs that matches the second block,
     evaluate the block."
```

```
public Y seek<Y,E>(Y seek(),
                   Y except(E e)) {
    try {
        return seek()
    }
    catch (E e) {
        return except(e)
    }
}
```

```
doc "Using the given resource, attempt to evaluate
     the first block."
public Y using<X,Y>(specified X resource,
                    Y seek(coordinated X x))
        given X satisfies Usable {
    try (resource) {
        return seek(resource)
    }
}
```

```
doc "Using the given resource, attempt to evaluate
     the first block. If an exception occurs that
     matches the second block, evaluate the second
     block."
public Y using<X,Y,E>(specified X resource,
                      Y seek(coordinated X x),
                      Y except(E e))
        given X satisfies Usable {
    try (resource) {
        return seek(resource)
    }
    catch (E e) {
        return except(e)
    }
}
```

The `lang.repetition` package defines support for loops.

```
doc "Repeat the block the given number of times."
public void repeat(Natural repetitions, void times()) {
    do(mutable Natural n:=0)
    while (n<repetitions) {
        times();
        n++;
    }
}
```

The `lang.quantification` package defines support for quantifiers.

```
doc "Count the elements for which the block
     evaluates to true."
public Natural count<X>(iterated Iterable<X> elements,
                        Boolean where(coordinated X x)) {
    mutable Natural count := 0;
    for (X x in elements) {
        if ( where(x) ) {
            ++count;
        }
    }
    return count
}
```

```
doc "Return true iff for every element, the block
     evaluates to true."
public Boolean forAll<X>(iterated Iterable<X> elements,
                         Boolean every(coordinated X x)) {
    for (X x in elements) {
        if ( !every(x) ) {
            return false
        }
    }
    return true
}
```

```
doc "Return true iff for some element, the block
     evaluates to true."
public Boolean forAny<X>(iterated Iterable<X> elements,
                         Boolean some(coordinated X x)) {
    Boolean where(X x) { return !some(x) }
```

```
    return !forAll(elements, where)
}
```

```
doc "Return the first element for which the block
     evaluates to true, or |null|if no such element
     is found."
public X? first<X>(iterated Iterable<X> elements,
                   Boolean where(coordinated X x)) {
    for (X x in elements) {
        if ( where(x) ) {
            return x
        }
    }
    return null
}
```

```
doc "Return the first element for which the first
     block evaluates to true, or the result of
     evaluating the second block, if no such
     element is found."
public X first<X>(iterated Iterable<X> elements,
                  Boolean where(coordinated X x),
                  X otherwise()) {
    if (exists X first = first(elements, where)) {
        return first
    }
    else {
        return otherwise()
    }
}
```

The `lang.comprehension.list` package defines support for `List` comprehensions.

```
doc "Iterate elements and return those for which
     the first block evaluates to true, ordered
     using the second block, if specified."
public List<X> from<X>(iterated Iterable<X> elements,
                       Boolean where(coordinated X x),
                       Comparable by(coordinated X x) = naturalOrder) {
    X select(X x) { return x }
    return from(elements, where, select, by)
}
```

```
doc "Iterate elements and for each element evaluate
     the first block. Build a list of the resulting
     values, ordered using the second block, if
     specified."
public List<Y> from<X,Y>(iterated Iterable<X> elements,
                         Y select(coordinated X x),
                         Comparable by(coordinated X x) = naturalOrder)  {
    Boolean where(X x) { return true }
    return from(elements, where, select, by)
}
```

```
doc "Iterate elements and select those for which
     the first block evaluates to true. For each of
     these, evaluate the second block. Build a list
     of the resulting values, ordered using the
     third block, if specified."
public List<Y> from<X,Y>(iterated Iterable<X> elements,
                         Boolean where(coordinated X x),
                         Y select(coordinated X x),
                         Comparable by(coordinated X x) = naturalOrder) {
    OpenList<Y> list = ArrayList<Y>();
    for (X x in elements) {
        if ( where(x) ) {
            list.append( select(x) );
        }
    }
    if (exists by) {
        list.sort(by);
    }
    return list
}
```

The `lang.comprehension.map` package defines support for `Map` comprehensions.

```
doc "Construct a |Map| by evaluating the block for
     each given key. Each |Entry| is constructed
```

```
     from the key and the value result of the
     evaluation."
public Map<U,V> mapFrom<U,V>(iterated Iterable<U> keys,
                             V to(coordinated U key)) {
    Entry<U,V> of(U key) { return key->to(key) }
    return map(keys, of)
}
```

```
doc "Construct a |Map| by evaluating the block for
     each given value. Each |Entry| is constructed
     from the value and the key result of the
     evaluation."
public Map<U,V> mapTo<U,V>(iterated Iterable<V> values,
                           U from(coordinated V value)) {
    Entry<U,V> of(V value) { return from(value)->value }
    return map(values, of)
}
```

```
doc "Construct a |Map| by evaluating the block for
     each given object and collecting the resulting
     |Entry|s."
public Map<U,V> map<X,U,V>(iterated Iterable<X> elements,
                           Entry<U,V> of(coordinated X element)) {
    OpenMap<U,V> map = HashMap<U,V>();
    for (X x in elements) {
        map.add( of(x) );
    }
}
```