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 now no longer the best foundation for solving 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.

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.

Ceylon features a similar inheritance and generic type model to Java. 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. Instead, like Scala, a type parameter may be marked as covariant or contravariant by the class or interface that declares the parameter. Ceylon supports *type aliases*, similar to C-style `typedef`.

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

Ceylon methods are similar to Java methods. Ceylon does not, strictly speaking, support first-class function types. However, higher-order functions are supported, with minimal extensions to the traditional C syntax. A method declaration may specify a *functional parameter* that accepts references to other methods with a certain signature. The argument of such a functional 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 method references. Finally, nested method declarations receive a closure of immutable values in the surrounding scope.

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. 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.

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

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.

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.

By default, Ceylon attributes and locals do not accept null values. Nullable locals and attributes must be explicitly declared using the `optional` annotation. Nullable 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 control flow structures are enhanced versions of the traditional constructs found in C, C# and Java. Even better, 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 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*.

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.

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.

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

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.

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.

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`.

1.1. A simple example

Here's a classic example, implemented in Ceylon:

```
doc "The classic Hello World program"
class Hello {
    log.info("Hello, World!");
}
```

This code defines a Ceylon class named `Hello`. When the class is instantiated, it calls the `info()` method of an attribute named `log` defined by the `lang.Object` class. (By default, this method displays its parameter on the console.) The `doc` annotation contains documentation that is included in the output of the Ceylon documentation compiler.

This improved version of the program takes a name as input from the console:

```
doc "A more personalized greeting"
class Hello(Process process) {
    String name = process.args.firstOrNull ? "World";
    log.info("Hello, ${name}!");
}
```

This time, the class has a parameter. The `Process` object has an attribute named `args`, which holds a `List` of the program's command line arguments. The local `name` is initialized from these arguments. The `?` operator returns the first argument that is not null. Finally, the value of the local is interpolated into the message string.

This looks a little cluttered. We can refactor the code and extract a method:

```
doc "A more personalized greeting"
class Hello(Process process) {

    sayHello( process.args.firstOrNull ? "World" );

    void sayHello(String name) {
        log.info("Hello, ${name}!");
    }

}
```

Now, lets rewrite the program as a web page:

```
import html.*;

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

    String name
        = request.parameters["name"].firstOrNull ? "World";

    head = Head { title="Hello World"; };

    body = Body {
```

```
        Div {
            cssClass = "greeting";
            "Hello, ${name}!"
        },
        Div {
            cssClass = "footer";
            "Powered by Ceylon."
        }
    };

}
```

This program demonstrates Ceylon's support for defining structured data, in this case, HTML. The `Hello` class extends Ceylon's `Html` class and initializes its `head` and `body` attributes. The `page` annotation specifies the URL at which this HTML should be accessible.

Chapter 2. Lexical structure

The lexical structure of Ceylon source files is very similar to Java.

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 in out void subtype def assign return break throw retry this super null none get set if else switch case for fail do while try catch finally exists nonempty is in public module package private abstract default override mutable optional static 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
```

```
1.
```

```
99E+3
```

2.4.2. Character literals

A single character literal consists of a character, preceded by a `@`:

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

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

```
EscapeSequence := "\" ("b" | "t" | "n" | "f" | "r" | "s" | "\" | """ | "'" | "$" | "{" | "}" )
```

The following are legal character literals:

```
@A
```

```
@#
```

```
@\n
```

2.4.3. String literals

A character string literal is a character sequence surrounded by double quotes, and may contain embedded expressions:

```
StringLiteral := """ ( StringCharacter+ | "${" Expression "}" )* """
```

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

The following are legal strings:

```
"Hello!"
```

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

```
"Hi there, ${name}!"
```

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
```

3.1.2. Imports and toplevel declarations

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

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

TODO: Should we support toplevel method declarations?

All toplevel types with a visibility modifier less strict than `private` must have the same name as the compilation unit filename (after removing the file suffix `.ceylon`). Unlike Java, a compilation unit may contain multiple toplevel class declarations with the same name.

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

```
Import* TypeDeclaration+
```

```
Import := "import" ImportPath ("." "*" | "alias" ImportElement)? ";"
```

```
ImportPath := ImportElement ("." ImportElement)*
```

```
ImportElement := PackageName | TypeName | MemberName
```

Each compilation unit belongs to exactly one *package*. A package is a namespace. Code in one compilation unit may refer to a toplevel type defined in another compilation unit in the same package without explicitly importing the type. It may refer to a toplevel type defined in a compilation in another package only if it explicitly imports the type.

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

An annotation is a `static` method invocation 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 declared as follows:

```
public multiplicity(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 `static` method:

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

Using a named parameter invocation of the `static` 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 (`static` methods) may return values of the same type.

3.1.4. Type

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

```
Type := RegularType | "subtype"
```

A type is a set of:

- attributes,
- non-`static` methods, and
- member classes.

A type may be *assignable* to another type. If `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`. The attribute is not `optional` unless the attribute of `Y` is `optional`.

- For each `mutable` attribute of `Y`, `X` has a `mutable` attribute with the same name and the same type. The attribute is `optional` if and only if the attribute of `Y` is `optional`.
- For each method of `Y`, `X` has a method with the same name, with the same formal parameter types, and whose return type is assignable to the return type of the method of `Y`. The method is not `optional` unless the method of `Y` is `optional`.
- For each member type of `Y`, `X` has a member type of the same name, with the same formal parameter types, that is assignable to the member type of `Y`.

Assignability obeys the following rules:

- Identity: `X` is assignable to `X`.
- Transitivity: if `X` is assignable to `Y` and `Y` is assignable to `Z` then `X` is assignable to `Z`.
- Single root: all types are assignable to `lang.Object`, including classes, interfaces, aliases and type parameters.

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 formal parameters.

```
RegularType := QualifiedTypeName TypeArguments?
```

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

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

A generic type must specify arguments for the generic type parameters.

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

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.

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

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

Every Ceylon class and 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).

```
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();
```

For a class declared `final`, any instance of the class may be assigned to the type `subtype`. For a class not declared `final`, only the `this` reference is assignable to `subtype`.

TODO: so this means we really need a `final` modifier after all? Is there another solution?

The type `subtype` is considered a covariant type parameter of the type, and may only appear in covariant positions of the type definition.

Should we support `Item[]` as a shorthand for `Sequence<Item>` or perhaps `List<Item>` and `Item[Key]` as a shorthand for `Correspondence<Item,Key>` or perhaps `Map<Item,Key>`? Should we support `Item...` as a shorthand for `Iterable<Item>`?

3.1.5. Generic type parameters

Methods, classes, interfaces and aliases may declare one or more generic type parameters. A class, interface or alias that declares type parameters is called a *parameterized type*. A method that declares type parameters is called a parameterized method.

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

A class or interface declaration with no type parameters defines exactly one type. A class or interface declaration with type parameters 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 method declaration with type parameters 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 class declaration with type parameters 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 assignable to every lower bound of the 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>
```

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 assignable to `T<A>`.

- If `X` is a contravariant type parameter, `T<A>` is assignable to `T<B>`.
- If `X` is nonvariant (neither covariant nor contravariant), there is no assignability 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.
- A lower bound of a type parameter in a covariant position is a contravariant position.
- A lower bound of a type parameter in a contravariant position is a covariant position.
- A covariant type parameter of a covariant position is a covariant position.
- A contravariant type parameter of a covariant position is a contravariant position.
- A covariant type parameter of a contravariant position is a contravariant position.
- A contravariant type parameter of a contravariant position is a covariant position.
- A nonvariant type parameter is a nonvariant position.
- A formal parameter of a functional parameter in a contravariant position is covariant.
- A formal parameter of a functional parameter in a covariant position is contravariant.
- The return type of a functional parameter in a contravariant position is contravariant.
- The return type of a functional parameter in a covariant position is covariant.

3.1.6. 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 *functional 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))
```

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. It must be a non-`optional` expression, unless the parameter is annotated `optional`.

A varargs parameter accepts a list of arguments or a single argument of type `Iterable`. Inside the method, it is available as a local of type `Iterable`.

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

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

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

TODO: should we just make `X...` a syntactic shorthand for `Iterable<X>` everywhere? Or, alternatively, should we also allow `Iterator<X>` to be passed to `X...`?

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)
```

An formal parameter may be annotated `optional`, in which case the parameter accepts a `null` argument.

Otherwise, if the `optional` annotation does not appear, the Ceylon compiler guarantees that the formal parameter never receives a `null` argument. Every argument of the formal parameter must be an expression of non-`optional` type.

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

3.1.7. Extended class

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

```
Supertype := "extends" RegularType 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 assignable to `Y`.

- If `X` extends `Y<B>`, then `X` is assignable to `Y<B>`.
- If `X<T>` extends `Y<T>`, then `X<B>` is assignable to `Y<B>` for any type `B`.
- If `X<T>` extends `Y`, then `X<B>` is assignable to `Y` for any type `B`.

3.1.8. 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 Sequence<T>, Collection<T>
```

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

- If `X` satisfies `Y`, then `X` is assignable to `Y`.
- If `X` satisfies `Y<B>`, then `X` is assignable to `Y<B>`.
- If `X<T>` satisfies `Y<T>`, then `X<B>` is assignable to `Y<B>` for any type `B`.
- If `X<T>` satisfies `Y`, then `X<B>` is assignable to `Y` for any type `B`.

3.1.9. Generic type constraint list

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

```
TypeConstraints = "where" TypeConstraint (AMPERSAND TypeConstraint)*
```

```
TypeConstraint := TypeName ( (">="|"<=") Type | "=" "subtype" | FormalParams )
```

There are four kinds of type constraints:

- an upper bound, `X >= T`, specifies that the type parameter `X` is assignable to a given type `T`,
- a lower bound, `X <= T`, specifies that a given type `T` is assignable to the type parameter `X`,
- a subtype bound, `X = subtype`, specifies that the type parameter `X` represents the concrete type of the current instance, and
- an initialization parameter specification, `X(...)` specifies that the type parameter `X` is a class with with the given formal parameter types.

Subtype bounds are needed since the special type `subtype` cannot appear in a contravariant position. A subtype bound cannot be applied to a covariant or contravariant 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 a lower bound must be a type which is assignable to the lower bounds.
- A type argument to a type parameter with an upper bound must be a type to which the upper bound is assignable.

A constraint affects the assignability of a type parameter:

- A type parameter is considered assignable to its lower bound.

- The upper bound of a type parameter is considered assignable to the type parameter.

Mutiple type constraints are seperated by `&`.

```
where X >= Number<X> & Y = subtype & Y(Natural count)
```

TODO: Should we use `for` instead of `where`?

TODO: Should we use `satisfies` instead of `>=`? For example `where X satisfies Number<X>`.

3.2. Classes

A *class* is a stateful, instantiable type. Classes are declared according to the following:

```
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,
- `static` method declarations and 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.

3.2.1. Class initializer

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

- The body of the class may declare *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); }
}
```

Class initialization parameters are optional. The following class:

```
public mutable class Point {
    public mutable Decimal x := 0.0;
    public mutable Decimal y := 0.0;
}
```

Is assumed to have zero initialization parameters.

TODO: is this the right thing to say? Alternatively, we could say that classes without a parameter list can't have statements or initialized simple attributes in the body of the class, and get the constructor automatically generated for the list of attributes.

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

```
public class SpecialKey1()
        extends Key( new 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 mutable class DiagonalPoint(Decimal position)
        extends Point() {

    x := y := sqrt(position**2/2) * position.sign;

    assert ("must have distance ${position} from origin")
        that ( x**2 + y**2 == position**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.

TODO: Does Ceylon support `static` member classes?

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.

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

3.2.2. Class inheritance

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

```
public mutable
class Customer(Name name, optional 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.

If a class does not explicitly specify a superclass using `extends`, its superclass is `lang.Object`.

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`.

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

3.2.3. 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 annotated `abstract` may not specify enumerated named instances.

A class with type parameters may not specify enumerated named instances.

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

```
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;

}
```

TODO: If we decide to support static attributes, then each `case` would be considered a static simple attribute.

A class with declared `cases` implicitly extends `lang.Selector`, a subclass of `java.lang.Enum`.

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

3.2.4. 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 different formal parameter types,
- 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.

3.2.5. 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 the same formal parameter types as the member class it overrides.

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.

TODO: If we decide to support `static` member classes, they cannot be overridden.

3.3. Interfaces

An *interface* is a type which does not specify the implementation of its members. Interfaces may not be directly instantiated. Interfaces are declared according to the following:

```
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.

3.3.1. Interface inheritance

An interface may extend any number of other interfaces.

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

The types listed after the `satisfies` keyword are the supertypes. All supertypes of an interface must be interfaces.

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

3.4. Methods

A *method* is a callable block of code. Methods may have parameters and may return a value. Methods are declared according to the following:

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

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

A method implementation may be specified using either:

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

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

TODO: Should we support toplevel methods instead of methods annotated `static`?

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.

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

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

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

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

```
public void addEntry(V key -> U value) { ... }
```

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

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

A method may declare multiple lists of parameters. Methods which declare more than one parameter list return 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.

A method reference returned by a method with multiple parameter lists must conform to the signature of the method with multiple parameter lists, after eliminating the first parameter list.

The Ceylon compiler preserves the names of method parameters, using a Java annotation.

A non-`void` method may be annotated `optional`, in which case the method may return a `null` value.

Otherwise, if the `optional` annotation does not appear, the Ceylon compiler guarantees that the method never returns `null`. Every `return` directive must specify an expression of non-`optional` type

The semantics of Ceylon methods are identical to Java, except that Ceylon methods may declare defaulted parameters and functional parameters.

3.4.1. Overloaded methods

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

- must have different formal parameter types, and
- may not declare default parameters.

Like Java, Ceylon resolves overloaded methods at compile time.

3.4.2. 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*.

Methods declared by interfaces never specify an implementation:

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

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.

3.4.3. Overriding methods

An abstract method, interface method, or a method annotated `default` may be overridden by subclasses of the class or interface which declares the method. The subclass must declare a method:

- annotated `override`,
- with the same name as the method it overrides,
- the same formal parameter types as the method it overrides,
- with a return type that is assignable to the return type of the method it overrides, and
- that is not annotated `optional`, unless the method it overrides is annotated `optional`.

Finally, the overridden methos class 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.

`static` methods cannot be overridden, and so `static` method invocation is never polymorphic.

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.

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.

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

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.

```
package mutable String firstName;
```

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

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

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

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

A simple attribute or attribute getter may be annotated `optional`, in which case the attribute may evaluate to `null`, and, if the attribute is also `mutable`, may have a null value assigned.

Otherwise, if the `optional` annotation does not appear, the Ceylon compiler guarantees that the attribute cannot evaluate to `null` or have a null value assigned. Every assignment to the attribute must specify an expression of non-`optional` type.

TODO: I would like to support `def` in place of the type for a block local attribute or local with an initializer or getter. For example:

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

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

```
mutable def count:=0;
```

3.5.1. Simple attributes and locals

A simple attribute defines state. Simple attributes are declared according to the following:

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

```
AttributeHeader := Annotation* Type MemberName
```

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.

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.

```
package mutable String firstName;
```

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

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

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.

3.5.2. Attribute getters

An attribute getter is declared as follows:

```
AttributeGetter := AttributeHeader Block
```

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

```
public Float total { return items.totalPrice; }
```

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

3.5.3. Attribute setters

An attribute setter is declared as follows:

```
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 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 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, and the attribute is not declared `optional`, the attribute is considered an *abstract attribute*.

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

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

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.

3.5.5. Overriding attributes

An abstract attribute, interface attribute, or attribute annotated `default` may be overridden by subclasses of the class or interface which declares the method. 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. The subclass 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, if the attribute it overrides is non-`mutable`, or with exactly the same type as the attribute it overrides, if the attribute it overrides is `mutable`,
- that is `mutable`, if the attribute it overrides is `mutable`,
- that is not `optional`, unless the attribute it overrides is `optional`, and
- that is `optional`, if the attribute it overrides is both `mutable` and `optional`.

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

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

TODO: if we decide to support `static` attributes, they can not be overridden.

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 any number of interfaces and at most one class.

The alias type is assignable to all of the satisfied types.

Any expression which is assignable to all the satisfied types is assignable to the alias type.

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

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

A shortcut is provided for definition of private aliases.

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

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.
- `static` specifies that a method can be called without an instance of the type that defines the method.
- `mutable` specifies that an attribute or local may be assigned, or that a class has assignable attributes.
- `optional` specifies that an attribute, local, parameter, or method return value may be null.
- `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.

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 Celon 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.

If two named declarations visible to some program element have the same name, and if they are not overloaded forms of the same class or method, then that name may not be used inside the program element unless:

- One declaration occurs in a block, class body, or interface body that is contained, directly or indirectly, inside the block, class body, or interface body that directly contains the other declaration. In this case, the name resolves to the inner declaration inside the block, class body, or interface body in which it occurs. We say that the inner declaration name *hides* the outer declaration name.
- One declaration is a class initialization parameter, and the other declares a member of the class. In this case the name resolves to the class initialization parameter inside the body of the class. We say that the initialization parameter name *hides* the class member name.

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 local;

    doc "The |public| visibility modifier
         annotation."
    public static oncePerElement Visibility public() {
        return public
    }

    doc "The |module| visibility modifier
```

```
         annotation."
    public static oncePerElement Visibility module() {
        return module
    }

    doc "The |package| visibility modifier
         annotation."
    public static oncePerElement Visibility package() {
        return package
    }

    doc "The |private| visibility modifier
         annotation."
    public static oncePerElement Visibility private() {
        return private
    }

}
```

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 `static`, non-`optional` method with exactly one formal parameter,
- a non-`static`, non-`optional` method with no formal parameters,
- a toplevel class with exactly one initialization parameter,
- a member class with no initialization parameters, or
- a non-`optional` attribute.

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

Extensions apply to a certain *extended type*:

- for static extension methods, the extended type is the declared type of the formal parameter,
- for non-`static` 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 class Person {
    ...
    public extension User user;
}
```

```
public static extension User personToUser(Person person) {
    return person.user;
}
```

```
public extension class CollectionUtils<T>(Collection<T> collection) {

```

```
    public Collection<T> nonZeroElements() {
        return collection.elements()
            having (T element) element!=0;
    }

    ...
}
```

Note: I actually much prefer the readability of `User personToUser(extends Person person)` and `CollectionUtils<T>(extends Collection<T> collection)`, but this doesn't work for attributes and non-`static` methods.

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

```
import org.domain.app.Extensions.personToUser;
import org.domain.utils.CollectionUtils;
```

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

An extension attribute, member class or non-`static` 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.Converters.personToUser;
...
Person person = ...;
User user = person;
```

```
import org.mydomain.myframework.CollectionUtils;
...
Collection<Integer> ints = ...;
Collection<Integer> result = ints.nonZeroElements();
```

An introduced type may result in an ambiguity:

- the extended type may have an attribute with the same name as an attribute of the introduced type, or
- the extended type may have a method with the same name and parameter types as a method of the introduced type.

TODO: What about ambiguities with multiple introduced types?

In this case, the attribute or method of the extended type is considered to override the method of the introduced type.

TODO: Or should we say the opposite?! What about using the `override` annotation to decide?

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.

3.7.4. Annotation constraints

The following annotations constrain the occurence of an annotation. By default, an annotation may appear multiple times on any program element.

- `inherited` specifies that the annotation is automatically inherited by subtypes.
- `multiplicity` specifies that the annotation may occur at most once in a certain scope. Its accepts one or more arguments of type `Multiplicity`: `onceEachElement`, `onceEachType`.
- `ofElements` specifies the kinds of program element at which the annotation occurs. Its accepts one or more arguments of type `Element`: `class`, `interface`, `method`, `attribute`, `alias`, `parameter`.
- `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 may only be applied to program elements at which the specified annotation occurs.

These annotations are applied to the static method declaration that defines an annotation, and the constraints are checked by the compiler.

```
public static ofElements(class) multiplicity(onceEachType)
Entity entity(LockMode lockMode) { return Entity(lockMode) }
```

```
public static ofElements(attribute, parameter) withType(String) multiplicity(onceEachElement)
Pattern pattern(Regex regex) { return Pattern(regex) }
```

TODO: Should there be an `annotation` modifier for static 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 class `doc.Documentation`:

```
public class Documentation {

    public static Description doc(String description) {
        return Description(description)
    }

    public static Sequence<Author> by(String... authors) {
        return from (String author in authors) select Author(author)
    }

    public static RelatedElement see(ProgramElement pe, String description=null) {
        return Related(pe, description)
    }

    public static ThrownException see(Type type<Exception>, String description=null) {
        return Related(type, description)
    }

}
```

TODO: should `see` and `throws` accept a list of `Entry` instead? For example:

```
see #Ruby->"if you are bored with Java"
    #Ceylon->"if you want to get some real work done"
```

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. 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 are executed sequentially.

```
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 between the declaration of a simple attribute or local and its first use in an expression. Each conditional branch must specify or assign a value to the simple attribute or local.

TODO: should we allow statements to be annotated, for example:

```
@doc "unsafe assignment" suppressWarnings(typesafety): apple = orange;
```

4.1.1. 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?

4.1.2. 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 | 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 `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
```

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, and may not be `optional` unless the method is annotated `optional`. When the directive is executed, the expression is evaluated 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 non-`optional` 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?
```

A `throw` directive may appear anywhere and may specify an expression of non-`optional` 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"
```

TODO: Should we have a `continue` or `redo` for the `do/while` loop, which lets you specify the next value of the iterator variable? This would let us get rid of `mutable` iterator variables, which are a minor wart.

TODO: We could support `for/fail` loops that return a value by supporting a `found expr` directive. We could support conditionals that return a value by adding a `then expr` directive.

4.1.3. 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 method reference 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, and may not be of `optional` type unless the attribute or local is declared `optional`.

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;
}
```

4.1.4. 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.

4.2. 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 `do/while` loop—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 | 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.

4.2.1. Control structure variables

Some control structures allow embedded declaration of a *variable* that is available inside the control structure body.

```
Variable := Type MemberName
```

A variable is treated as a formal parameter of the control structure body.

TODO: Should we say that type declarations in control structure variables are optional? The compiler should be able to infer the type of the variable.

4.2.2. 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 `optional` type evaluates to any value other than `null`,
- a *nonemptiness condition* is satisfied when an expression of type `optional Container` evaluates to a non-empty instance, and
- a *subtype condition* is satisfied when an expression evaluates to an instance of a specified type.

```
Condition := Expression | ExistsCondition | IsCondition
```

```
ExistsCondition := ("exists" | "nonempty") (Variable Specifier | Expression)
```

```
IsCondition := "is" (Variable Specifier | Type Expression)
```

Any condition contains an expression. In the case of existence, nonemptiness and subtype 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 non-`optional` type `Boolean`.
- If the `is` modifier appears, the condition expression must be of non-`optional` type.
- If the `exists` modifier appears, the condition expression must be of `optional` type.
- 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 may be declared without the `optional` annotation, even though the specifier expression is of `optional` type. The type of the specifier expression must be assignable to the declared type of the variable.
- If the condition expression is a local, the local will be treated by the compiler as having non-`optional` type inside the block that follows immediately, relaxing the usual compile-time restrictions upon `optional` types.

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 the specified type when the control structure is executed.
- If the `exists` modifier appears, the condition is satisfied if the expression evaluates to a non-null value when the control structure is executed.
- If the `nonempty` modifier appears, the condition is satisfied if the expression evaluates to a non-null, non-empty instance of `Container` when the control structure is executed.

4.2.3. `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(optional 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;
}
```

4.2.4. `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 `Case<X>`, where `X` is the `switch` expression type, or an explicit `null`. Alternatively, the `case`s may be subtype conditions.

```
Case := Expression ("," Expression)* | "is" Type | "null"
```

If a `case (null)` occurs, the `switch` expression must be of `optional` type.

If the `switch` expression is of `optional` type, there must be an explicit `case (null)`.

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 a subtype 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 ...)`?

4.2.5. `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 := Variable ("->" Variable)? "in" Expression
```

The type of the iterated expression depends upon the iterarion variable declarations:

- The iterated expression must be an expression of type assignable to `Iterable<X>` or `Iterator<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>>` or `Iterator<Entry<U,V>>` where `U` and `V` are the declared types of the iteration variables.

When the construct is executed:

- if the iterated expression is of type `Iterable`, the `Iterator` is obtained by calling `iterator()`, and then
- the `for` block is executed once for each element of the `Iterator`.

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); }
```

```
for (String key -> Natural value in map) {
    log.info("${key} = ${value}");
}
```

```
for (Person p in people) {
    log.debug("found ${p.name}");
    if (p.age >= 18) {
        log.info("found an adult: ${p.name}");
        break true
    }
}
fail {
    log.info("no adults");
}
```

4.2.6. `do/while`

The `do/while` loop has the form:

```
DoWhile := Do? While
```

```
Do := "do" ("(" DoIterator ")")? Block?
```

```
While := "while" "(" Condition ")" (";" | Block)
```

The loop may declare an iterator variable of any type, which may be `mutable`.

```
DoIterator := Annotation* Variable (Specifier | Initializer)
```

When the construct is executed, both blocks are executed repeatedly, until the termination condition first evaluates to `false`, at which point iteration ends. In each iteration, the first block is executed before the condition is evaluated, and the second block is executed after it is evaluated.

TODO: does `do/while` need a fail block? Python has it, but what is the real usecase?

For example:

```
do (mutable Natural n:=0) {
    log.info("count = " + $n);
}
while (n<=10) { n++; }
```

```
do (Iterator<Person> iter = org.employees.iterator())
while (iter.more) {
    log.info( iter.next().name );
}
```

```
do (Iterator<Person> iter = people.iterator())
while (iter.more) {
    Person p = iter.next();
    log.debug(p.name);
    p.greet();
}
```

```
mutable Person person := ....;
while (exists Person parent = person.parent) {
    log.info(parent.name);
    person := parent;
}
```

```
mutable Person person := ....;
do {
    log.info(person.name);
    person := person.parent;
}
while (!person.dead);
```

```
do (mutable Person person := ...) {
    log.info(person.name);
}
while (!person.parent.dead) {
    person := person.parent;
}
```

TODO: `do/while` is significantly enhanced compared to other Java-like languages. Is this truly a good thing?

4.2.7. `try/catch/finally`

The `try/catch/finally` exception manager has the form:

```
TryCatchFinally := Try Catch* Finally?
```

```
Try := "try" Resources? Block
```

```
Catch := "catch" "(" Variable ")" Block
```

```
Finally := "finally" Block
```

The type of each `catch` variable must be assignable to `lang.Exception`.

The `try` block may declare a list of *resource* expressions. A resource expresson produces a heavyweight object that must be released when execution of the `try` terminates.

```
Resources := "(" Resource ("," Resource)* ")"
```

Each resource expression must be assignable to `lang.Usable`.

```
Resource := Variable Specifier | Expression
```

When the construct is executed:

- each resource expression is evaluated, and then `begin()` is called upon each resulting resource instance, then
- the `try` block is executed, then
- `end()` is called upon each resource instance, with the exception that propagated out of the `try` block, if any, then
- if an exception did propagate out of the `try` block block, the first matching `catch` block, 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) { map[key] := value; }
```

(This example shows the Ceylon version of Java's `synchronized` keyword.)

```
try ( Transaction(), Session s = Session() ) {
    Person p = s.get(#Person, id)
    ...
    return p
}
catch (NotFoundException e) {
    return null
}
```

The `retry` directive re-evaluates the resource expressions, and then re-executes the `try` block, calling `begin()` and `end()` upon each resource.

```
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,
- method 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 | SpecialValue | Enumeration | "(" Expression ")"
```

A *primary* is formed by recursively invoking or evaluating members of an atom or static members of a type.

```
Primary := Atom | EnumeratedInstanceReference | MethodReference | Invocation | Evaluation
```

More complex expressions are formed by combining expressions using operators, including assignment operators.

```
Expression := Primary | Assignment | OperatorExpression | Meta
```

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), and
- the nullability of the expression (whether it can evaluate to `null`).

If an expression can evaluate to `null`, we say it is of `optional` type. Otherwise, we say it is of non-`optional` type. For a nullable expression of type `T` we often say that its type is `optional T`.

An expression is assignable to a program element if:

- the type of the expression is assignable to the declared type of the program element, and
- the expression is non-`optional`, or the program element is declared `optional`.

TODO: We need to allow some way to have an expression to be specified as a toplevel element, for DSLs.

5.1. 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`.

```
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.

All literal values are expressions of non-`optional` type.

5.1.1. 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;
```

5.1.2. Floating point number literals

A floating point number literal is an expression of type `lang.Float`.

```
public static Float pi = 3.14159;
```

5.1.3. Character literals

A single character literal is an expression of type `lang.Character`.

```
if ( string[i] == @+ ) { ... }
```

TODO: do we really need character literals??

5.1.4. String literals

A character string literal is an expression of type `lang.String`.

```
person.name := "Gavin";
```

```
log.info("${Time()} ${message}");
```

```
String multiline = "Strings may
span multiple lines
if you prefer.";
```

```
display("Melbourne\tVic\tAustralia\nAtlanta\tGA\tUSA\nGuanajuato\tGto\tMexico\n");
```

TODO: we need to support that the embedded expressions are evaluated lazily when the string literal appears as a method parameter. This is important for log messages and assertions. So should `"${Time()} ${message}"` actually be a method of type `String()` with no parameters, instead of a `String`? (And we would have a built-in converter to do implicit conversion to `String`.)

TODO: is `"{Time()} {message}"` a better syntax for interpolation?

5.1.5. 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 = email.matches( '^\w+@((\w+)\.)+$' );
```

```
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 embedded expressions using the `${...}` escape sequence, 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'`.

5.2. `this`, `super`, `null` and `none`

The type of the following special values depends upon the context in which they appear.

```
SpecialValue := "this" | "super" | "null" | "none"
```

The keyword `null` refers to a special value that is assignable to all `optional` types, and never to non-`optional` types. The Ceylon language and compiler ensures that this value never receives any invocation.

The keyword `super` refers to the current instance (the instance that is being invoked), and is of the same type as the immediate superclass of the class. It is an expression of non-`optional` type. 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.

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. It is an expression of non-`optional` type.

The keyword `none` refers to a special value that is assignable to `Sequence`. It is an expression of non-`optional` type. This value has no elements.

5.3. Metamodel references

The `Type` object for a type, the `Method` object for a method, or the `Attribute` object for an attribute may be referred to using a special syntax.

```
Meta := TypeMeta | MemberMeta
```

```
TypeMeta := HASH RegularType
```

```
MemberMeta := HASH (RegularType ".")? MemberName
```

Metamodel references are compile-time typesafe.

```
Type<List<String>> stringListType = #List<String>;
```

```
Attribute<Person, String> nameAttribute = #Person.name;
```

```
Method<Person, String>> sayMethod = #Person.say;
```

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?

5.4. Enumerated instance references

An enumerated instance of a class is identified according to:

```
EnumeratedInstanceReference := (RegularType ".")? MemberName
```

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;
```

An enumerated instance reference is an expression of non-`optional` type.

5.5. Method references

A *method reference* is an invocable reference to an instance method, static method, attribute getter or setter or type constructor:

```
MethodReference :=
InstanceMethodReference | StaticMethodReference |
AttributeReference |
ConstructorReference | MemberConstructorReference
```

An ordinary method reference is a reference to an instance or static method.

A *constructor reference* is a reference to an implicit method with a signature that depends upon the formal parameters of the class. For a class `T`, the constructor method has non-`optional` return type `T` and the same formal parameters as the class.

A *getter or setter method reference* is a reference to an implicit method with a signature that depends upon the attribute type. For an attribute of type `T`:

- The getter method has no formal parameters, return type `T`, and is of `optional` type if and only if the attribute is declared `optional`.
- If the attribute is `mutable`, the `void` setter method has a single formal parameter of type `T` of `optional` type if and only if the attribute is declared `optional`.

5.5.1. Static method and toplevel class constructor references

A constructor reference for a toplevel type consists of a named type.

```
ConstructorReference := RegularType
```

A static method reference consists of a named type and named method of that type.

```
StaticMethodReference := (RegularType ".")? MemberName
```

TODO: According to this, we can "curry" in type arguments of the type. Is this right? If so, why can't we curry type arguments of the member?

5.5.2. Instance method, attribute getter and setter, and member class constructor references

Instance method references, attribute getter and setter references, and member class constructor references specify a name of a member, an invocation operator, and a *receiver expression* that evaluates to an instance or `Iterable` set of instances of a type that has a member with that name.

Otherwise, if no receiver expression is explicitly specified, the current object is the receiver.

When the method reference expression is executed, the receiver expression is evaluated and a reference to the resulting value is held with the method reference.

```
Receiver := (Primary InvocationOperator)?
```

```
InstanceMethodReference := Receiver MemberName
```

```
MemberConstructorReference := Receiver RegularType
```

```
AttributeReference := ("set" | "get")? Receiver MemberName
```

The keyword `get` or `set` in an attribute reference determines if it is the attribute getter or setter that is referred to.

TODO: Here are a couple of alternative syntax possibilities for attribute getter/setter references:

- *`get(foo.bar)` and `set(foo.bar)`*
- *`foo.get:bar` and `foo.set:bar`*
- *`foo.bar get` and `foo.bar set`*

5.5.3. Invocation operators

References to instance methods, attributes, and member classes use one of three different invocation operators.

```
InvocationOperator := "." | "?." | "*."
```

The invocation operator depends upon the type of the receiver expression. If `X` is the type that has the instance method, attribute, or member type, the invocation operator must be:

- `.` if the expression is of non-`optional` type `X`,
- `.?` if the expression is of type `optional X`, or
- `.*` if the expression is of non-`optional` type `Iterable<X>`.

5.5.4. Specifying a method reference as a method implementation or functional parameter argument

A method reference may be used to define a method using `=`.

```
Comparison order(X x, Y y) = Order.reverse;
```

```
void display(String message) = log.info;
```

```
String newString(Character... chars) = String;
```

Method references may appear as an argument to a functional parameter, either as a positional argument, or as an argument specified using = in a named parameter invocation.

This method has a functional parameter:

```
void sort(List<String> list, Comparison by(String x, String y)) { ... }
```

This code passes a reference to a local method with a conforming signature to the method.

```
Comparison reverseAlpha(String x, String y) { return y<=>x; }
sort(names, reverseAlpha);
```

This class has two functional parameters:

```
public class TextInput(Natural size=30, String onInit(), void onUpdate(String s)) { ... }
```

This code instantiates the class, passing references to the getter/setter method pair of an attribute:

```
TextInput it = TextInput {
    size=15;
    onInit = get person.name;
    onUpdate = set person.name;
}
```

Method references do not, strictly speaking, have types. However, the signature of a method reference determines if it can appear to the right of the = specifier in a method declaration, or as an argument to a functional parameter. In this case we say the method reference *conforms* to the signature of the method or functional parameter.

A method reference *conforms* to a method or functional parameter signature if:

- the return type of the method reference is assignable to the declared return type of the method or functional parameter,
- the method reference is of non-`optional` return type, unless the method or functional parameter is declared `optional` type, and
- the method reference has exactly the same number of formal parameters, with the same types, as the method or functional parameter.

Thus, method signatures are covariant in return type, and nonvariant in parameter types.

TODO: could we make method references contravariant in the parameter types? If so, we should use the same rule for overriding.

5.5.5. Returning a method reference from a method

A method reference may be returned by a method with multiple parameter lists. The reference must conform to the signature of the method, after elimination of the first parameter list.

```
Comparison getOrder(Boolean reverse=false)(Natural x, Natural y) {
    Comparison natural(Natural x, Natural y) { return x<=>y; }
    Comparison reverse(Natural x, Natural y) { return y<=>x; }
    if (reverse) {
        return reverse;
    else {
        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);
```

Of course, more than one argument list may be specified in a single expression:

```
Comparison comp = getOrder(true)(10, 100);
```

5.6. Invocation

Methods references are *invokable*. An *invocation* consists of an expression that evaluates to a method reference, together with an argument list.

```
Invocation := Primary TypeArguments? Arguments
```

An invocation must specify arguments for parameters, either by listing or naming parameter values.

```
Arguments := PositionalArguments FunctionalArguments? | NamedArguments
```

Arguments to required parameters must be specified by the caller. Arguments to defaulted parameters and varargs may optionally be specified.

For a required or defaulted formal parameter of declared type `T`, the type of the argument expression must be assignable to `T`. The argument expression must be of non-`optional` type, unless the formal parameter is annotated `optional`.

For a varargs parameter of declared type `T...`, there may either:

- be a single argument expression of non-`optional` type assignable to `Iterable<T>`, or
- an arbitrary number of argument expressions of non-`optional` type assignable to `T`.

In the second case, the argument expressions are evaluated and collected into an instance of `Iterable<T>` when the invocation is executed.

When an invocation is executed:

- each argument is evaluated in turn in the calling context, then
- the receiving instance, if any, and the name of the invoked member or type is determined by evaluating the receiver expression and obtaining a method reference, and
- 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.

5.6.1. 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.

TODO: What does a `void` method invocation evaluate to? The receiving instance?

```
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); }
```

```
HashCode.calculate(default, firstName, initial, lastName)
```

```
HashCode.calculate { algorithm=default; firstName, initial, lastName }
```

```
from (people) having (Person p) (p.age>18) select (Person p) (p.name);
```

```
iterate (map) perform (String name->Object value) { log.info("${name} ${value}"); };
```

The type of a method invocation expression depends upon the invocation operator. If `X` is the declared return type of the method:

- If the invocation operator is `.`, the expression is of type `X`. It is of `optional` type if and only if the method is declared `optional`.
- If the invocation operator is `?.`, the expression is of type `optional X`.
- If the invocation operator is `*.`, the expression is of non-`optional` type `Sequence<X>`.
- If no receiver expression is specified, the expression is of type `X`. It is of `optional` type if and only if the method is declared `optional`.

5.6.2. 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);
        }
    }
}
```

The type of a method invocation expression depends upon the invocation operator. If `X` is class:

- If the invocation operator is `.`, the expression is of non-`optional` type `X`.
- If the invocation operator is `?.`, the expression is of type `optional X`.
- If the invocation operator is `*.`, the expression is of non-`optional` type `Sequence<X>`.
- If no receiver expression is specified, the expression is of non-`optional` type `X`.

5.6.3. 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)
```

5.6.4. 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 functional parameters, specifies the type, a formal parameter list and a block of code (an inline method declaration).

```
NamedArgument := SpecifiedNamedArgument | FunctionalNamedArgument
```

```
SpecifiedNamedArgument := ParameterName Specifier ";"
```

```
FunctionalNamedArgument := (Type | "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; }
}
```

TODO: Getter, setter specification for parameters declared `mutable`?

TODO: should we allow a comma-separated list of values to be specified here, thereby allowing the braces around a sequence enumeration to be omitted in this case?

5.6.5. Vararg arguments

Vararg arguments are seperated by commas.

```
VarargArguments := VarargArgument ("," VarargArgument)*
```

```
VarargArgument := Expression | Variable 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 an `Iterable` of the parameter type.

```
( {1, 1, 2, 3, 5, 8} )
```

5.6.6. 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; }
```

5.6.7. Functional arguments

After a positional argument list, arguments to functional parameters may be specified with certain punctuation eliminated:

- if the functional parameter declares no formal parameters, 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 *functional arguments* of a positional parameter invocation.

```
FunctionalBody := FormalParameters? ( Block | "(" Expression ")" )
```

```
FunctionalArguments := (ParameterName FunctionalBody)+
```

For example:

```
having (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); }
```

Functional arguments are listed without any additional punctuation:

```
ifTrue (x+1) ifFalse (x-1)
```

```
select (Person p) (p.name) having (Person p) (p.age>18)
```

Arguments must be listed in the same order as the formal parameters are declared by the method declaration.

5.6.8. Iteration

A specialized invocation syntax is provided for static methods which iterate collections. If the first parameter of the method is of type `Iterable<X>`, annotated `iterated`, and all remaining parameters are functional parameters, then the method may be invoked according to the following protocol:

```
StaticMethodReference "(" ForIterator ")" FunctionalArguments
```

And then if a functional 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 static
List<Y> from<X,Y>(iterated Iterable<X> elements,
                  Boolean having(coordinated X x),
                  Y select(coordinated X x));
```

We may invoke the method as follows:

```
List<String> names = from (Person p in people) having (p>20) select (p.name);
```

Which is equivalent to:

```
List<String> names = from (people) having (Person p) (p>20) select (Person p) (p.name);
```

Or, we may invoke the method as follows:

```
List<String> labels = from (Key key -> Value value in namedValues)
                      having (user.authorized(key))
                      select ("${key} ${value}");
```

Which is equivalent to:

```
List<String> labels = from (namedValues)
                      having (Key key -> Value value) (user.authorized(key))
                      select (Key key -> Value value) ("${key} ${value}");
```

5.6.9. Variable definition

A specialized invocation syntax is also provided for static methods which define a variable. If the first parameter of the method is of type `X`, annotated `specified`, and all remaining parameters are functional parameters, then the method may be invoked according to the following protocol:

```
StaticMethodReference "(" Variable Specifier ")" FunctionalArguments
```

And then if a functional 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 static
Y ifExists<X,Y>(specified optional 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 static
Y using<X,Y>(specified X resource,
             Y seek(coordinated X x) )
  where X >= 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));
```

5.6.10. Resolving overloaded methods and types

For invocations of overloaded methods, and instantiation of overloaded types, the specific overloaded declaration is resolved by the Ceylon compiler at compile time.

Overloaded declarations are resolved by considering the *erased signature* of the method or type declarations.

The erased signature of a type is a obtained by taking the declared parameter types and:

- if type arguments were explicitly specified by the instantiation, substituting the type arguments explicitly specified for type parameters that appear in the parameter types, or, otherwise,
- if type arguments were not explicitly specified, substituting the lower bound of the type parameter for each type parameter that appears in the parameter types (or `lang.Object` if the type parameter has no lower bound).

The erased signature of a method is obtained by taking the declared parameter types and:

- first, substituting type arguments of the receiving type for the corresponding type parameter wherever it appears in the method parameter types, and then
- if type arguments were explicitly specified by the invocation, substituting the type arguments explicitly specified for method type parameters that appear in the parameter types, or, otherwise,
- if type arguments were not explicitly specified, substituting the lower bound of the type parameter for each method type parameter that appears in the parameter types (or `lang.Object` if the type parameter has no lower bound).

An invocation or instantiation resolves to a particular overloaded method or type declaration if there is exactly one overloaded declaration of the named method or type with an erased signature to which the given arguments expression types are assignable.

If more than one overloaded declaration has an erased signature to which the arguments are assignable, or if there is no declaration with an erased 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.)

Note that method references are not resolved to a specific overloaded declaration of the method or type except in the context of an invocation of the method reference. However, the compiler does check that an appropriate overloaded declaration exists before permitting specification of the method reference in the declaration of another method or functional argument.

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 signature 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 surrounding 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.)

5.7. Enumeration

A sequence may be instantiated by enumerating the elements:

```
Enumeration := "{" ( Expression ("," Expression)* )? "}"
```

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 superype of all the enumerated element expression types. The concrete type of this object is not specified here.

```
Sequence<String> names = { "gavin", "max", "emmanuel", "steve", "christian" };
```

Empty braces `{}` and `none` are synonyms for a special value that can be assigned to `Sequence`.

```
OpenList<Connection> connections = {};
```

An enumeration is an expression of non-`optional` type.

There are no true literals for 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" };
```

```
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 = {};
```

This representation is used as the canonical literal form for collections.

5.8. Evaluation

An attribute evaluation consists of an attribute name, an evaluation operator, and a *receiver expression* that evaluates to an instance or `Iterable` set of instances of a type that has an attribute with that name.

If no receiver expression is explicitly specified, the current object is the receiver.

Evaluation of a local is a special case. There is no receiving instance or evaluation operator.

```
Evaluation := (Primary EvaluationOperator)? MemberName
```

```
EvaluationOperator := "." | "?." | "*."
```

The evaluation operator depends upon the type of the receiver expression. If `X` is the type that has the attribute, the evaluation operator must be:

- `.` if the expression is of non-`optional` type `X`,
- `.?` if the expression is of type `optional X`, or
- `.*` if the expression is of non-`optional` type `Iterable<X>`.

When an attribute evaluation is executed:

- the receiving instance is determined by evaluating the receiver expression, and
- 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 method or initializer ends without a thrown exception, execution of the calling context resumes.

The value of the attribute evaluation is the current value of the simple attribute or the return value of the attribute getter, as specified by the `return` directive.

```
String name = person.name;
```

The value of the local evaluation is the current value of the local.

The type of an attribute invocation expression depends upon the evaluation operator. If `X` is the declared type of the attribute:

- If the evaluation operator is `.`, the expression is of type `X`. It is of `optional` type if and only if the attribute is `optional`.
- If the evaluation operator is `?.`, the expression is of type `optional X`.
- If the evaluation operator is `*.`, the expression is of non-`optional` type `Sequence<X>`.
- If no receiver expression is specified, the expression is of type `X`. It is of `optional` type if and only if the attribute is `optional`.

The type of a local evaluation expression is the type of the local. It is of `optional` type if and only if the local is `optional`.

5.9. Assignment

Certain expressions are *assignable*. An assignable expression may appear to the left of the `:=` (assign) operator, and possibly, depending upon the type of the expression, to the left of the numeric, logical or bitwise compound assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `&&=`, `||=`, `?=` or as the subject of the increment and decrement operators `++`, `--`, or apply operator `.=`.

For example, the following expressions are assignable:

- a local declared `mutable`, for example `count := 0`,
- any attribute expression where the underlying attribute has a setter or is a simple attribute declared `mutable`, for example `person.name := "Gavin"`,
- element expressions for the type `OpenCorrespondence`, for example `order.lineItems[0] := lineItem`, and
- range expressions for the type `OpenSequence`, for example `fibonacci[0..1] := {0,1}`.

5.9.1. Assignable expressions

An assignable expression consists of an attribute name, and, optionally, in the case of an attribute of type `OpenCorrespondence`, a specification of the element keys to be assigned to, together with a receiver expression that evaluates to an instance of a type with an attribute with that name.

If no receiver expression is explicitly specified, the current object is the receiver.

Assignment to a local is a special case. There is no receiving instance.

```
AssignableExpression := (Primary ".")? MemberName ElementSpec?
```

When an assignment expression or unary assignment expression is executed, the new value to be assigned is computed, the receiving object is obtained by evaluating the receiver expression, the actual member to be invoked is determined by considering the runtime type of the receiving instance, and then:

- if the assignable expression is a simple attribute or local, the value of the simple attribute of local is set to the new value,
- if the assignable expression is an attribute with a setter, execution of the calling context pauses while the body of the attribute setter is executed by the receiving instance with the new value, and then, when execution of the setterends without a thrown exception, execution of the calling context resumes, or
- if the assignable expression is an attribute of type `OpenCorrespondence` together with an element key specification, the key expression is evaluates, and the `define()` method is invoked with the resulting key and the new value.

5.9.2. Assignment

An assignment consists of an assignable expression, an assignment operator and an expression to be assigned.

```
Assignment := AssignableExpression AssignmentOperator Expression
```

```
AssignmentOperator :=
":=" |
NumericAssignmentOperator |
BitwiseAssignmentOperator |
LogicalAssignmentOperator |
DefaultAssignmentOperator
```

```
NumericAssignmentOperator := "+=" | "-=" | "*=" | "/=" | "%="
```

```
BitwiseAssignmentOperator := "&=" | "|=" | "^="
```

```
LogicalAssignmentOperator := "&&=" | "||="
```

```
DefaultAssignmentOperator := "?="
```

The assignment operator might be `:=`, which simply evaluates the expression to be assigned and assigns the resulting value to the assignable expression, or it might be a compound assignment operator, which evaluates both expressions, computes a single value from the two resulting values, and assigns this value to the assignable expression.

This statement sets the value of an attribute:

```
person.name := "Gavin";
```

This statement assigns to a keyed element of a `Sequence`, resulting in a call to the `define()` method of `OpenCorrespondence`:

```
order.lineItems[0] := LineItem { product = prod; quantity = 1; };
```

The following rules must be satisfied:

- the assignable expression must be a `mutable` attribute with no element key specification, or an attribute of type `OpenCorrespondence` together with an element key specification,
- the receiver expression must not be of `optional` type,
- the type of the expression to be assigned must be assignable to the type of the assignable expression, and
- if the expression to be assigned is of `optional` type, then the assignable expression must be of `optional` type,

TODO: fix to account for range assignment!

The whole assignment construct is itself an expression. The type of the assignment expression is the type of the assignable expression. The assignment expression is of `optional` type if and only if the expression to be assigned is of `optional` type.

When the assignment is executed, the assignment expression evaluates to the new value that was assigned.

5.9.3. Unary assignment

Unary increment and decrement and the apply aperator `.=` are also considered a kind of assignment, called *unary assignment*.

```
IncrementOrDecrement := PrefixIncrementOrDecrement | PostfixIncrementOrDecrement | Apply
```

```
PrefixIncrementOrDecrement := IncrementOrDecrementOperator AssignableExpression
```

```
PostfixIncrementOrDecrement := AssignableExpression IncrementOrDecrementOperator
```

```
IncrementOrDecrementOperator := "++" | "--"
```

```
Apply := AssignableExpression ".=" MemberName Arguments?
```

Unary assignment evaluates the assignable expression, computes a new value by applying an operation to the result, and then assigns this new value to the assignable expression.

The following rules must be satisfied:

- the assignable expression must be a `mutable` attribute with no element key specification, or an attribute of type `OpenCorrespondence` together with an element key specification,
- the receiver expression must not be of `optional` type, and
- the assignable expression must not be of `optional` type.

The whole unary assignment construct is itself an expression. The type of the unary assignment expression is the type of the assignable expression. The unary assignment expression is of non-`optional` type.

When the assignment is executed, the unary assignment expression evaluates to the new value that was assigned, except in the case of postfix increment or decrement, in which case the unary assignment expression evaluates to the previous value of the assignable expression, before the unary assignment was executed.

5.10. 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;
```

```
++count;
```

```
Integer j = i++;
```

```
if ( x > 100 ) { ... }
```

```
User gavin = users["Gavin"];
```

```
List<Item> firstPage = list[0..20];
```

```
for ( Natural n in 1..10 ) { ... }
```

```
List<Day> nonworkDays = days[{0,7}];
```

```
if ( name == value ) return ... ;
```

```
log.info( "Hello " + $person + "!")
```

```
List<String> names = { person1, person2 }*.name;
```

```
optional String name = person?.name;
```

This table defines operator precedence from highest to lowest, along with associativity rules:

Table 5.1.

Operations	Operators	Type	Associativity
Member invocation and lookup, subrange,	`.`, `*.`, `?.`, `(,,)`, `{;;;}`, `[]`, `?[]`, `[..]`,	Binary / ternary /	Left

Operations	Operators	Type	Associativity
postfix increment, postfix decrement:	`[...]`, `++`, `--`	N-ary / unary postfix	
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 `**`.

The following tables define the semantics of the Ceylon operators:

5.10.1. 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	Equivalent	LHS type	RHS type	Return type
Member invocation						
`.`	`lhs.member`	invoke		`X` or type `X`	Member of `X`	Member type
`?.`	`lhs?.member`	nullsafe invoke	`if (exists lhs) lhs.member else null`	`optional X`	Member of `X`	`optional` member type
`*.`	`lhs*.member`	spread invoke	`for (X x in lhs) x.member`	`Iterable<X>`	Member of	`Sequence` of

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
					`X`	member type
Assignment						
`:=`	`lhs := rhs`	assign	`Object.assign(set lhs,rhs)`	`X` or `optional X`	`X` or `optional X`	`X` or `optional X`
Render						
`$`	`$rhs`	render	`if (exists rhs) rhs.string else ""`		`optional Object`	`String`
Compound invocation assignment						
`.=`	`lhs.=member`	apply	`Object.assign(set lhs,lhs.member)`	`X`	Member of `X`, of type `X`	`X`
Type or method argument specification						
`(,,)` or `{;;;}`	`lhs(x,y,z)` or `lhs { a=x; b=y; c=z; }`	arguments		Type or method	Parameter types of type or method	Type or return type of method

5.10.2. Equality and comparison operators

These operators compare values for equality, order, magnitude, or membership, producing boolean values.

Table 5.3.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Equality						
`===`	`lhs === rhs`	identical	`Object.identical(lhs, rhs)`	`optional Object`	`optional Object`	`Boolean`
`==`	`lhs == rhs`	equal	`if (exists lhs) lhs.equals(rhs) else if (exists rhs) false else true`	`optional Object`	`optional Object`	`Boolean`
`!=`	`lhs != rhs`	not equal	`if (exists lhs) lhs.equals(rhs).complement else if (exists rhs) true else false`	`optional Object`	`optional Object`	`Boolean`
Comparison						
`<=>`	`lhs <=> rhs`	compare	`lhs.compare(rhs)`	`Comparable<T>`	`T`	`Comparison`
`<`	`lhs < rhs`	smaller	`lhs.compare(rhs).smaller`	`Comparable<T>`	`T`	`Boolean`
`>`	`lhs > rhs`	larger	`lhs.compare(rhs).larger`	`Comparable<T>`	`T`	`Boolean`
`<=`	`lhs <= rhs`	small as	`lhs.compare(rhs).smallAs`	`Comparable<T>`	`T`	`Boolean`
`>=`	`lhs >= rhs`	large as	`lhs.compare(rhs).largeAs`	`Compar-`	`T`	`Boolean`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
				`able<T>`		
			Containment			
`in`	`lhs in rhs`	in	`lhs.in(rhs)`	`Object`	`Category` or `Iterable<Object>`	`Boolean`
			Assignability			
`is`	`lhs is Rhs`	is	`lhs.instanceOf(#Rhs)`	`Object`	Any type	`Boolean`

TODO: should we really have the equality operators accept null values?

5.10.3. Logical operators

These are the usual logical operations for boolean values.

Table 5.4.

Op	Example	Name	Equivalent	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 Object.assign(set lhs,rhs)`	`Boolean`	`Boolean`	`Boolean`
`&&=`	`lhs &&= rhs`	conditional and	`if (lhs) Object.assign(set lhs,rhs) else false`	`Boolean`	`Boolean`	`Boolean`

5.10.4. Operators for handling null values

These operators make it easy to work with `optional` types.

Table 5.5.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
			Existence			
`exists`	`lhs exists`	exists	`if (exists lhs) true else false`	`optional Object`		`Boolean`
`nonempty`	`lhs nonempty`	nonempty	`if (exists lhs) lhs.empty.complement else false`	`optional Container`		`Boolean`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Default						
`?`	`lhs ? rhs`	default	`if (exists lhs) lhs else rhs`	`optional T`	`T` or `optional T`	`T` or `optional T`
Default assignment						
`?=`	`lhs ?= rhs`	default assignment	`if (exists lhs) lhs else Object.assign(set lhs,rhs)`	`optional T`	`T` or `optional T`	`T` or `optional T`

5.10.5. 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	Equivalent	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.value(index) else null`	`optional Correspondence<X,Y>`	`X`	`optional Y`
`[]`	`lhs[indices]`	list lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`Sequence<X>`	`Sequence<Y>`
`[]`	`lhs[indices]`	set lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`Iterable<X>`	`Iterable<Y>`
Sequence subranges						
`[..]`	`lhs[x..y]`	subrange	`Sequences.range(lhs,x,y)`	`S where S>=Sequence<X> & S(X...elements)`	Two `Natural` values	`S`
`[...]`	`lhs[x...]`	upper range	`Sequences.range(lhs,x)`	`S where S>=Sequence<X> & S(X...elements)`	`Natural`	`S`
Sequence concatenation						
`+`	`lhs + rhs`	join	`Sequences.join(lhs, rhs)`	`S where S>=Sequence<X> & S(X...elements)`	`S`	`S`

TODO: Should we overload `*` for sequences, like in some other languages? It is nice to be able to do stuff like `"-"*n`.

TODO: Should we have operators for set union/intersection/complement and set comparison?

TODO: Apparently there is an issue with correctly lexing `[1..2]` and `[1...]`, so we might need to go with `[1,2]` and `[1,.]` or `[1:2]` and `[1:.]` instead.

5.10.6. Operators for constructing objects

These operators simplify the syntax for constructing certain commonly used built-in types.

Table 5.7.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Range and entry constructors						
`..`	`lhs .. rhs`	range	`Range(lhs, rhs)`	`T where T >= Ordinal & T >= 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?

TODO: Apparently there is an issue with correctly lexing `1..2`, so we might need to go with `[1,2]` or `[1:2]` instead.

5.10.7. Arithmetic operators

These are the usual mathematical operations for all kinds of numeric values.

Table 5.8.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Increment, decrement						
`++`	`++rhs`	successor	`Object.assign(set rhs,rhs.successor)`		`Ordinal<T>`	`T`
`--`	`--rhs`	predecessor	`Object.assign(set rhs,rhs.predecessor)`		`Ordinal<T>`	`T`
`++`	`lhs++`	increment	`(Object.assign(set lhs,lhs.successor)).predecessor`	`Ordinal<T>`		`T`
`--`	`lhs--`	decrement	`(Object.assign(set lhs,lhs.predecessor)).successor`	`Ordinal<T>`		`T`
Numeric operations						
`-`	`-rhs`	negation	`rhs.inverse`		`Numeric<N,I>`	`I`
`+`	`lhs + rhs`	sum	`lhs.plus(rhs)`	`Numeric<N,I>`	`N`	`N`
`-`	`lhs - rhs`	difference	`lhs.minus(rhs)`	`Numeric<N,I>`	`N`	`I`
`*`	`lhs * rhs`	product	`lhs.times(rhs)`	`Numeric<N,I>`	`N`	`N`
`/`	`lhs / rhs`	quotient	`lhs.divided(rhs)`	`Numeric<N,I>`	`N`	`N`
`%`	`lhs % rhs`	remainder	`lhs.remainder(rhs)`	`Integral<N,I>`	`N`	`N`
`**`	`lhs ** rhs`	power	`lhs.power(rhs)`	`Numeric<N,I>`	`N`	`N`

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Numeric assignment						
`+=`	`lhs += rhs`	add	`Object.assign(set lhs,lhs.plus(rhs))`	`Numeric<N,I>`	`N`	`N`
`-=`	`lhs -= rhs`	subtract	`Object.assign(set lhs,lhs.minus(rhs))`	`Numeric<N,I>`	`N`	`I`
`*=`	`lhs *= rhs`	multiply	`Object.assign(set lhs,lhs.times(rhs))`	`Numeric<N,I>`	`N`	`N`
`/=`	`lhs /= rhs`	divide	`Object.assign(set lhs,lhs.divided(rhs))`	`Numeric<N,I>`	`N`	`N`
`%=`	`lhs %= rhs`	remainder	`Object.assign(set lhs,lhs.remainder(rhs))`	`Integral<N,I>`	`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`.

5.10.8. 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	Equivalent	LHS type	RHS type	Return type
Bitwise operations						
`~`	`~rhs`	complement	`rhs.complement`		`Bits<X>`	`X`
`|`	`lhs | rhs`	or	`lhs.or(rhs)`	`Bits<X>`	`Bits<X>`	`X`
`&`	`lhs & rhs`	and	`lhs.and(rhs)`	`Bits<X>`	`Bits<X>`	`X`
`^`	`lhs ^ rhs`	exclusive or	`lhs.xor(rhs)`	`Bits<X>`	`Bits<X>`	`X`
Bitwise assignment						
`|=`	`lhs |= rhs`	or	`Object.assign(set lhs,lhs.or(rhs))`	`Bits<X>`	`Bits<X>`	`X`
`&=`	`lhs &= rhs`	and	`Object.assign(set lhs,lhs.and(rhs))`	`Bits<X>`	`Bits<X>`	`X`
`^=`	`lhs ^= rhs`	exclusive or	`Object.assign(set lhs,lhs.xor(rhs))`	`Bits<X>`	`Bits<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 root type

The `lang.Object` class is the root of the type hierarchy and supports the binary operators `==` (equals), `!=` (not equals), `===` (identity equals), `.` (invoke), `in` (in), `is` (is), the unary prefix operator `$` (render), and the binary operator `:=` (assign).

In addition, references of type `optional Object` support the binary operators `?.` (nullsafe invoke) and `?` (default) and the unary operator `exists`, along with the binary operators `==` (equals), `!=` (not equals), `===` (identity equals) and `:=` (assign).

```
public abstract class Object {

    doc "The equals operator x == y. Default implementation compares
         attributes annotated |id|, or performs identity comparison."
    see #id
    public 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 Integer hash { return ... }

    doc "Computes the hash code of the instance using the given
         attributes."
    public Integer hash(Attribute... attributes) { ... }

    doc "The unary render operator $x. 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 String string { return ... }

    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 belongs to the given |Iterable|
         object or listed objects."
    see #Iterable
    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 "Binary assignability operator x is Y. Determine if the
         instance is of the given |Type|."
    public Boolean instanceOf(Type<Object> type) {
        return this.type.assignableTo(type)
    }

    doc "A log obect for the type."
    public static Log log = Log(type);

    ...

}
```

TODO: is this really the root type in Ceylon, or do we need some other type sitting above `lang.Object` and `java.lang.Object`, to accommodate classes from other languages?

6.2. Boolean values

The `lang.Boolean` class represents boolean values, supports the binary `||, &&` and `=>` operators and unary `!` operator and

inherits 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)
    }
}
```

6.3. 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);

}
```

Classes with enumerated `cases` implicitly extend `lang.Selector`

```
public abstract class Selector(String name, int ordinal)
        satisfies Case<subtype> { ... }
```

6.4. 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);

}
```

6.5. Iterable objects and iterators

The interface `lang.Container` represents the abstract notion of an object that may be empty. It supports the unary postfix operator `nonempty`.

```
public interface Container {

    doc "The nonempty operator. Determine
         if the container is empty."
    public Boolean empty;

}
```

The `lang.Iterable<X>` interface represents a type that may be iterated over using a `lang.Iterator<X>`. It supports the binary operator `*.` (spread).

```
public interface Iterable<out X> satisfies Container {

    doc "Produce an iterator."
    public Iterator<X> iterator();

}
```

```
public interface Iterator<out X> {

    doc "The status of the iterator. |true|
         indicates that the iterator contains
         more elements."
    public Boolean more;

    doc "The current element."
    public X current;

    doc "Advance to the next element, returning
         the next element."
    throw #ExhaustedIteratorException
          "if the iterator contains no
           more elements."
    public X next();

}
```

An iterator is used according to the following idiom:

```
do ( Iterator i = list.iterator() )
while (i.more) {
    doSomething( i.next() );
}
```

Some iterable objects may support element removal during iteration.

```
public mutable interface OpenIterable<X>
        satisfies Iterable<X> {

    public override OpenIterator<X> iterator();

}
```

```
public mutable interface OpenIterator<X>
        satisfies OpenIterator<X> {

    doc "Remove the current element from the iterable
         object to which this iterator belongs."
    public void remove();
```

```
    doc "Replace the current element in the iterable
         object to which this iterator belongs with
         the given object."
    public assign current;

}
```

6.6. Categories

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 {

    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);

}
```

6.7. 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> {

    doc "Binary lookup operator x[key]. Returns the value defined
         for the given key."
    throws #UndefinedKeyException
           "if no value is defined for the given key"
    public V value(U 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);

}
```

A decorator allows retrieval of lists and sets of values.

```
public extension class Correspondences<in U, out V>(Correspondence<U, V> correspondence) {

    doc "Binary list lookup operator x[keys]. Returns 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 List<V> values(List<U> keys) {
        return from (U key in keys) select (correspondence.lookup(key))
    }

    doc "Binary set lookup operator x[keys]. Returns a set of
         values defined for the given set of keys."
    throws #UndefinedKeyException
           "if no value is defined for one of the given keys"
    public Set<V> values(Set<U> keys) {
        return ( from (U key in keys) select (correspondence.lookup(key)) ).elements
    }

}
```

TODO: should we handle the list lookup and set lookup operators using a helper class that calls a constructor of the concrete subtype, like we do with `Sequence`?

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> {

     doc "Element assignment operator x[key] = value. Assign a value
          to the given key. Return the previous value for the key,
          or null if there was no value defined."
     public optional V define(U key, V value);

}
```

A decorator allows addition of multiple `Entry`s.

```
public extension class OpenCorrespondences<in U, V>(OpenCorrespondence<U, V> correspondence) {

     doc "Add the given entries, overriding any definitions that
          already exist."
     public void define(Entry<U, V>... definitions) {
         for (U key->V value) {
             correspondence.define(key, value);
         }
     }

}
```

6.8. Sequences

A `lang.Sequence` is a correspondence from a bounded progression of natural numbers. Sequences support the binary operators `+` (join), `[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, out X>, Iterable<X> {

    doc "The index of the last element of the sequence."
    throws #EmptyException
           "if the sequence has no elements"
    public Natural lastIndex;

}
```

A helper class defines the join and range operators, for sequence types which have an appropriate constructor, and decorates `Sequence` with some convenience attributes:

```
public extension class Sequences<out X>(Sequence<X> sequence) {

    doc "The binary join operator x + y.
         The returned sequence does not reflect changes
         to the original sequences."
    public static S join<S,T>(S... sequences)
            where S>=Sequence<T> & S(T... elements) {

        class JoinedIterable<T>
            implements Iterable<T> { ... }

        return S( JoinedIterable() );
    }

    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 static S range<S,T>(S sequence, Natural from, Natural to=sequence.lastIndex)
             where S>=Sequence<T> & S(T... elements) {

        class SubrangeIterable<T>
            implements Iterable<T> { ... }

        return S( SubrangeIterable() );
    }
```

```
    doc "The first element of the sequence."
    throws #EmptyException
           "if the list is empty"
    public X first {
        if (sequence.empty) {
            throw EmptyException();
        }
        else {
            return sequence[0];
        }
    }

    doc "The last element of the sequence."
    throws #EmptyException
           "if the list is empty"
    public X last {
        return sequence[sequence.lastIndex];
    }

    doc "The first element of the sequence, or null if
         the sequence is empty."
    public optional X firstOrNull {
        if (sequence.empty) {
            return first;
        }
        else {
            return null;
        }
    }

    doc "The last element of the sequence, or null if
         the sequence is empty."
    public optional X lastOrNull {
        if (sequence.empty) {
            return last;
        }
        else {
            return null;
        }
    }

}
```

There is a mutable subtype which allows assignment to an index.

```
public mutable interface OpenSequence<X>
        satisfies Sequence<X>, OpenCorrespondence<Natural,X> {}
```

6.9. 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, #key, #value)
    }

    override public Integer hash {
        return hash(#key, #value)
    }

}
```

6.10. Collections

The interface `lang.Collection` is the root of the Ceylon collections framework.

```
public interface Collection<out X>
        satisfies Iterable<X>, Category {

    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 having(X element));

    doc "Determine if the given condition evaluates to |true|
         for at least one element or entry."
    public Boolean contains(Boolean having(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 having(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) where T <= X;

    doc "A mutable copy of the collection."
    public OpenCollection<T> copy<T>() where T <= X;

}
```

A decorator provides the ability to sort collections of `Comparable` values in natural order.

```
public extension class CollectionsOfComparable<out X>(Collection<X> collection)
        where X>=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 OpenIterable<X>, OpenCategory<X>, Collection<X> {

    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 having(X element));

}
```

6.10.1. Sets

Sets implement the following interface:

```
public interface Set<out X>
        satisfies Collection<X>, Correspondence<Object, Boolean> {
```

```
    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) where T <= X;
    public override OpenSet<T> copy<T>() where T <= X;

}
```

There is a mutable subtype:

```
public mutable interface OpenSet<X>
        satisfies Set<X>, OpenCollection<X>, OpenCorrespondence<Object, Boolean> {}
```

6.10.2. Lists

Lists implement the following interface, and support the operators inherited from `Collection` and `Sequence`:

```
public interface List<out X>
        satisfies Collection<X>, Sequence<X> {

    doc "The tail of the list. The returned list does
         reflect changes to the original list."
    public List<X> rest;

    doc "The index of the first element of the list
         which satisfies the condition."
    throws #NotFoundException
           "if no element satsfies the condition"
    public Natural firstIndex(Boolean having(X element));

    doc "The index of the last element of the list
         which satisfies the condition."
    throws #NotFoundException
           "if no element satsfies the condition"
    public Natural lastIndex(Boolean having(X element));

    doc "The index of the first element of the list
         which satisfies the condition, or null if
         no element satisfies the condition."
    public optional Natural firstIndexOrNull(Boolean having(X element));

    doc "The index of the last element of the list
         which satisfies the condition, or null if
         no element satisfies the condition."
    public optional Natural lastIndexOrNull(Boolean having(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 "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) where T <= 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) where T <= 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>() where T <= X;

}
```

It is possible to iterate lists without creating an iterator:

```
do ( mutable List<Person> people := ... )
while (people nonempty) {
    doSomething(people.first);
    people.=rest;
}
```

There is a mutable subtype:

```
public mutable interface OpenList<X>
        satisfies List<X>, OpenCollection<X>, OpenSequence<X> {

    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."
    public X removeIndex(Natural at);

    doc "Add the given elements at the end of the list."
    public void prepend(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 append(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."
    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."
    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> rest;
    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)
        where X>=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);
    }

}
```

6.10.3. Maps

Maps implement the following interface:

TODO: is it OK that maps are not contravariant in U?

```
public interface Map<U, out V>
        satisfies Collection<Entry<U,V>>, Correspondence<U, V> {

    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 "Return the value defined for the given key, or null
         if no value is defined for that key."
    public optional V valueOrNull(U key);

    doc "Produce a new map by applying an operation to every
         element of the map."
    public Map<U, W> transform<W>(optional 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 having(U key -> V value));

    public override Map<U, T> with<T>(Entry<U, T>... entries) where T <= V;
    public override OpenMap<U,T> copy<U,T>() where T <= V;

}
```

There is a mutable subtype:

```
public mutable interface OpenMap<U,V>
        satisfies Map<U,V>, OpenCollection<Entry<U,V>>, OpenCorrespondence<U, V> {

    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 having(U key));

    override public OpenSet<U> keys;
    override public OpenBag<V> values;
    override public OpenMap<V, Set<U>> inverse;

}
```

6.10.4. Bags

Bags implement the following interface:

```
public interface Bag<out X>
        satisfies Collection<X>, Correspondence<Object, Natural> {

    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) where T <= X;
    public override OpenBag<T> copy<T>() where T <= 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;

}
```

6.10.5. Collection operations

```
public Collections {

    public static List<X> join<X>(List<X> list...) {
        return new List<X> {
            ...
        }
    }

    public static Bag<X> union<X>(Bag<X> bag...) {
        return new Bag<X> {
            ...
        }
    }

    public static Set<X> union<X>(Set<X> set...) {
        return new Set<X> {
            ...
        }
    }

    public static Set<X> intersection<X>(Set<X> set...) {
        return new Set<X> {
            ...
        }
    }

    public static Set<X> complement<X>(Set<X> set, Set<Object> sets...) {
        return new Set<X> {
            ...
        }
    }

}
```

6.11. Ordered values

The `lang.Comparable` interface represents totally ordered types, and supports the binary operators `>`, `<`, `<=`, `>=` and `<=>` (compare).

```
public interface Comparable<in T> {

    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 `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;

}
```

6.12. 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 Sequence<X>, Case<Object>
        where X>=Ordinal & X>=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 |List| 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 Sequence<X> by(Natural stepSize) { return ... }

}
```

6.13. 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 extension Character parseUtf16Character() { return ... }

        doc "Parse the string representation of a |Character| in UTF-8"
        public extension Character parseUtf8Character() { return ... }

    }

}
```

Strings implement `Sequence`, therefore they support the join, subrange, contains and lookup operators, among others.

```
public class String(Character... characters)
        satisfies Comparable<String>, Sequence<Character>, 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 "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(Sequence<Character> whitespace = " \n\l\r\t") { return ... }

    doc "Collapse substrings of the given characters into
         single space characters."
    public String normalize(Sequence<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`?

6.14. 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 Iterator<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?

6.15. Bit strings

The interface `Bits` represents a fixed length string of boolean values, and supports the binary `|, &, ^` and unary `~` operat-

ors.

```
public interface Bits<T>
        where T = subtype {

    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;

}
```

6.16. 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 `+`, `-`, `*`, `/`, `**`, and the unary prefix operators `-`, `+`. In addition, `mutable` values of type `lang.Numeric` support the compound assignment operators `+=`, `-=`, `/=`, `*=`.

```
public interface Numeric<N, I>
        satisfies Number, Comparable<N>
        where I >= Number & N = subtype {

    doc "The unary - operator"
    public I inverse;

    doc "The binary + operator"
    public N plus(N number);

    doc "The binary - operator"
    public I 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);

}
```

The subtype `lang.Integral` supports the binary operator `%`, and inherits the unary operators `++` and `--` from `Ordinal`.

```
public interface Integral<N, I>
        satisfies Numeric<N, I>, Ordinal
        where I >= Number & N = subtype {

    doc "The binary % operator"
    public N remainder(N number);

}
```

TODO: should `plus()` and `times()` accept varargs, to minimize method calls?

Five numeric types are built in:

`lang.Natural` represents 63 bit unsigned integers (including zero).

```
public class Natural(Natural natural)
        satisfies Integral<Natural,Integer>, Case<Integral>, 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,Integer>, Case<Integral> {
    ...

    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,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,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,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 ... }

    }

}
```

6.17. 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,
        optional Natural seconds=null, optional Natural milliseconds=null,
        optional Timezone timezone=null)
        extends Instant() {
    public Natural hours = hours;
    public Natural minutes = minutes;
    public optional Natural seconds = seconds;
    public optional Natural milliseconds = milliseconds;
    public optional 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)
        where X >= Instant {
    public X start = start;
    public X end = end;
    public Duration<X> duration { return ...; }
    ...
}
```

```
public class Duration<X>(Map<Granularity<X>, Natural> magnitude)
        where X >= 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>
        where X >= 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;
}
```

6.18. Control expressions

The `lang` package defines several classes containing static methods for building complex expressons.

```
public class 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 static void assert(String message(), Boolean that()) {
        if ( assertionsEnabled() && !that() ) {
            throw new AssertionException( message() )
        }
    }

}
```

```
public class Conditionals {

    doc "If the condition is true, evaluate first block,
         and return the result. Otherwise, return a null
         value."
    public static optional 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 static 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 a null
         value."
    public static optional Y ifExists<X,Y>(specified optional 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 static Y ifExists<X,Y>(specified optional X value,
                                  Y then(coordinated X x),
```

```
                                  Y otherwise()) {
        if (exists value) {
            return then(value)
        }
        else {
            return otherwise()
        }
    }

    doc "Evaluate the block which matches the selector value, and
         return the result of the evaluation. If no block matches
         the selector value, return a null value."
    public static optional Y select<X,Y>(X selector,
                                         cases Iterable<Entry<Case<X>, functor Y()>> value) {
        for (Case<X> match -> X evaluate() in value) {
            if ( match.test(selector) ) {
                return evaluate(value)
            }
        }
        return null;
    }

    doc "Evaluate the block which matches the selector value, and
         return the result of the evaluation. If no block matches
         the selector value, evaluate the last block and return
         the result of the evaluation."
    public static Y select<X,Y>(X selector,
                                cases Iterable<Entry<Case<X>, functor Y()>> value,
                                Y otherwise()) {
        if (exists Y y = select(selector, value)) {
            return y
        }
        else {
            return otherwise()
        }
    }

}
```

```
public class Loops {

    doc "Repeat the block the given number of times."
    public static void repeat(Natural repetitions, void times()) {
        do(mutable Natural n:=0)
        while (n<repetitions) {
            times();
            n++;
        }
    }

}
```

```
public class Quantifiers {

    doc "Count the elements for with the block evaluates to true."
    public static Natural count<X>(iterated Iterable<X> elements,
                                   Boolean having(coordinated X x)) {
        mutable Natural count := 0;
        for (X x in elements) {
            if ( having(x) ) {
                ++count;
            }
        }
        return count;
    }

    doc "Return true iff for every element, the block evaluates to true."
    public static 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 static Boolean forAny<X>(iterated Iterable<X> elements,
                                    Boolean some(coordinated X x)) {
        return !forAll(elements, (X x) !some(x));
    }

    doc "Return the first element for which the block evaluates to true,
```

```
         or a null value if no such element is found."
    public static optional X first<X>(iterated Iterable<X> elements,
                                      Boolean having(coordinated X x)) {
        for (X x in elements) {
            if ( having(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 static X first<X>(iterated Iterable<X> elements,
                             Boolean having(coordinated X x),
                             X otherwise()) {
        if(exists X first = first(elements, having)) {
            return first
        }
        else {
            return otherwise()
        }
    }

}
```

```
public class ListComprehensions {

    doc "Iterate elements and return those for which the first
         block evaluates to true, ordered using the second block,
         if specified."
    public static List<X> from<X>(iterated Iterable<X> elements,
                                  Boolean having(coordinated X x),
                                  optional Comparable by(coordinated X x) = null) {
        return from(elements, having, (X x) x, 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 static List<Y> from<X,Y>(iterated Iterable<X> elements,
                                    Y select(coordinated X x),
                                    optional Comparable by(coordinated X x) = null)  {
        return from(elements, (X x) true, 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 static List<Y> from<X,Y>(iterated Iterable<X> elements,
                                    Boolean having(coordinated X x),
                                    Y select(coordinated X x),
                                    optional Comparable by(coordinated X x) = null) {
        OpenList<Y> list = ArrayList<Y>();
        for (X x in elements) {
            if ( having(x) ) {
                list.append( select(x) );
            }
        }
        if (exists by) {
            list.sort(by);
        }
        return list
    }

}
```

```
public class MapComprehensions {

    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 static Map<U,V> mapFrom<U,V>(iterated Iterable<U> keys,
                                        V to(coordinated U key)) {
        return map(keys, (U key) key->to(key))
    }

    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 static Map<U,V> mapTo<U,V>(iterated Iterable<V> values,
                                      U from(coordinated V value)) {
```

```
        return map(values, (V value) from(value)->value)
    }

    doc "Construct a |Map| by evaluating the block for each given object
         and collecting the resulting |Entry|s."
    public static 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) );
        }
    }

}
```

```
public class Handlers {

    doc "Attempt to evaluate the first block. If an exception occurs that
         matches the second block, evaluate the block."
    public static 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 static Y using<X,Y>(specified X resource,
                               Y seek(coordinated X x))
                    where X >= Usable {
        try (resource) {
            return seek(resource)
        }
    }

    doc "Using the given resource, attempt to evaluate the first block.
         If an exception occurs that matches the block, evaluate the
         second block."
    public static Y using<X,Y,E>(specified X resource,
                                 Y seek(coordinated X x),
                                 Y except(E e))
                    where X >= Usable {
        try (resource) {
            return seek(resource)
        }
        catch (E e) {
            return except(e)
        }
    }

}
```

6.19. Primitive type optimization

For certain types, the Ceylon compiler is permitted to transform local declarations to Java primitive types, literal values to Java literals, and operator invocations to use of native Java operators, as long as the transformation does not affect the semantics of the code.

For this example:

```
Integer calc(Integer j) {
    Integer i = list.size;
    i++;
    return i * j + 1000;
}
```

the following equivalent Java code is acceptable:

```
Integer calc(Integer j) {
    int i = list.size().get();
    i++;
    return new Integer( i * lang.Util.intValue(j) + 1000 );
}
```

The following optimizations are allowed:

- `lang.Boolean` to Java `boolean`
- `lang.Natural` to Java `long`
- `small lang.Natural` to Java `int`
- `lang.Integer` to Java `long`
- `small lang.Integer` to Java `int`
- `lang.Float` to Java `double`
- `small lang.Float` to Java `float`

`lang.Character` may not be optimized to the Java `char` type, since Java `char` represents a UTF-16 character.

However, these optimizations may never be performed for locals, attributes or method types declared `optional`.

The following operators may be optimized: `+, -, *, /, ++, --, +=, -=, *=, /=, >, <, <=, >=, ==, &&, ||, !`.

Finally, integer, float and boolean literals may be optimized.