Project Ceylon

A Better Java

Version: For internal discussion only

Table of Contents

A work in progress

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 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 improve compile-time typesafety using special handling for null values,
- to provide language-level modularity, and
- to improve on Java's very primitive facilities for meta-programming, thus making it easier to write frameworks.

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 methods are similar to Java methods. However, Ceylon does not feature any Java-like constructor declaration and so each Ceylon class has exactly one "constructor". As an alternative to overloading, Ceylon supports method and constructor parameters with default values. Ceylon does not support first-class function types, but references to methods may be passed to other methods and methods may declared inline in an invocation.

Ceylon classes do not contain fields, in the traditional sense. Instead, Ceylon supports only a higher-level construct: *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 must be assigned when the class is instantiated. An immutable local may not be assigned more than once.

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.

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. Unlike Java and C#, Ceylon features language-level support for dates and times.

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#, but 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. One application of this syntax is the support for Java/C#-like code annotations. 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.

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 four levels of visibility for program elements: private (the default), `package`, `module` and `public`. There's no equivalent of 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 private 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. Declarations

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

2.1. General declaration syntax

All declarations follow a general pattern.

2.1.1. 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 and locals must be named with an initial lowercase or underscore.

```
PackageName := LIdentifier
```

```
TypeName := UIdentifier
```

```
MemberName := LIdentifier
```

```
ParameterName := LIdentifier
```

2.1.2. Type declarations

A type declaration declares a class, interface or type alias.

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

TODO: Should we support toplevel static method declarations?

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

```
Import* TypeDeclaration+
```

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

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

2.1.3. 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.

An annotation is a static method call. For an annotation with no arguments, or with only literal-valued arguments, the parentheses around, and commas between, the positional arguments may be omitted.

For example:

```
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
```

2.1.4. 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 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. For example:

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

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

Defaulted parameters specify a default value.

```
DefaultParam := FormalParam Specifier
```

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

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

For example:

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

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
```

For example:

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

TODO: should we just make `X...` a syntactic shorthand for `Iterable<X>` everywhere?

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)
```

2.1.5. Generic type parameters

Method, class and interface declarations may declare generic type parameters.

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

```
TypeParam := Variance TypeName
```

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

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

For example:

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

```
Sender<in M>
```

```
Container<out T>
```

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

2.1.6. Type declaration

Method and attribute declarations must declare a type.

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

Most types are classes or interfaces:

```
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)* ">"
```

For example:

```
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).

For example:

```
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`.

The type `subtype` is considered a covariant type parameter of the type, and may not appear in contravariant positions of the type declaration.

2.1.7. Extended class

Classes may extend other classes using the `extends` clause.

```
Supertype := "extends" RegularType PositionalArguments
```

For example:

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

2.1.8. Satisfied interfaces

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

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

For example:

```
satisfies Sequence<T>, Collection<T>
```

2.1.9. 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:

- upper bounds,
- lower bounds
- subtype bounds, and
- initialization parameter specifications.

For example:

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

Subtype bounds are needed since the special type `subtype` cannot appear in a contravariant position.

Initialization parameter specifications allow instantiation of the generic type.

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

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

2.2. 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? DeclarationOrStatement* "}"
```

2.2.1. Class inheritance

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

```
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. The semantics of class inheritance are exactly the same as Java, and the above declarations are equivalent to the following Java declarations:

```
public class Customer
        extends Person {
    public Customer(Name name) { this(name, null); }
    public Customer(Name name, Organization org) { super(name, org); }
    ...
}
```

```
class Token
        extends Datetime
        implements Comparable<Token>, Identifier {
    public Token() { super(); }
    ...
}
```

2.2.2. Class instantiation

Ceylon classes do not support a Java-like constructor declaration syntax. However, Ceylon supports *class initialization parameters*. A class initialization parameter may be used anywhere in the class body. The class body is executed every time the class is instantiated.

All non-`optional` attributes of the class must be explicitly initialized somewhere in the body of the class definition, unless the class is declared `abstract`, in which case they must be initialized within the body of every subclass definition.

This declaration:

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

Is equivalent to this Java class:

```
public class Key {
    private final ReadAttribute<Lock> lock;
    public ReadAttribute<Lock> lock() { return lock; }

    public Key(Lock lock) {
        this.lock = new SimpleReadAttribute<Lock>(lock);
    }

}
```

This declaration:

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

Is equivalent to this Java class:

```
public class Key {
    private Lock _lock;

    private final ReadAttribute<Lock> lock = new ReadAttribute<Lock>() {
        @Override public Lock get() { return _lock; }
    };
    public ReadAttribute<Lock> lock() { return lock; }

    public Key(Lock lock) {
        _lock = lock;
    }

}
```

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 equivalent to this Java class with a default constructor:

```
public class Point {
    private final Attribute<Decimal> x = new SimpleAttribute<Decimal>( new Decimal(0.0) );
    public SimpleAttribute<Decimal> x() { return x; }

    private final Attribute<Decimal> y = new SimpleAttribute<Decimal>( new Decimal(0.0) );
    public SimpleAttribute<Decimal> y() { return y; }
}
```

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.

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

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

Which are equivalent to the Java:

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

```
public class SpecialKey2
        extends Key {
```

```
    public SpecialKey2(Lock lock) {
        super(lock);
    }
    ...
}
```

The body of a class may contain arbitrary code, which is executed when the class is instantiated.

```
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;

}
```

The compiler is permitted to optimize private attribute declarations. So the above class is equivalent to:

```
public class DiagonalPoint extends Point {

    public DiagonalPoint(final Decimal position) {

        x = y = sqrt( position.power(2).divided(2) ).times(position.sign);

        assert_(new F0<String>() {
                   public String call() {
                       return "must have distance " + position + " from origin";
                   }
                },
                new F0<Boolean>() {
                   public Boolean call() {
                       return ( x.power(2) + y.power(2) ).equals( position.power(2) );
                   }
                });
    }

}
```

TODO: should class initialization parameters be allowed to be declared `mutable`?

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

2.2.3. Defaulted parameters

When a class with a defaulted parameter is instantiated, and a value is not assigned to the defaulted parameter by the caller, the default value defined by the specifier is used.

This class:

```
public class Counter(Natural initialCount=0) { ... }
```

Is equivalent to a class with three Java constructor declarations and an inner class:

```
public class Counter {

    public Counter() {
        Counter(0);
    }

    public Counter(Natural initialCount) {
        ...;
    }

    public Counter(CounterParameters namedParameters) {
        Counter( namedParameters.initialCount );
    }

    public static class CounterParameters {
        private Natural initialCount=0;
        CounterParameters initialCount(Natural initialCount) {
            this.initialCount = initialCount;
            return this;
        }
    }

}
```

This named parameter call:

```
Counter { initialCount=10; }
```

Is equivalent to this Java code:

```
new Counter ( new CounterParameters().initialCount(10) );
```

2.2.4. Annotations

Every annotation is a static (non-void) method call. This Ceylon class:

```
doc "Represents a person"
by "Gavin"
public class Person { ... }
```

Is equivalent to this Java code:

```
public class Person { ...

    static {
        Type<Person> type = Type.get(Person.class);
        type.addAnnotation( doc("Represents a person") );
        type.addAnnotation( by("Gavin") );
        type.addAnnotation( public() );
    }

    ...

}
```

2.2.5. Class instance enumeration

The keyword `case` is used to specify an enumerated named instance of a class. All `cases` 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 `;` instead of `...`, additional instances of the class may not be instantiated.

```
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 `case`s implicitly extends `lang.Selector`, a subclass of `java.lang.Enum`. The above declarations are equivalent to the following Java declarations:

```
public class DayOfWeek
        extends Selector<DayOfWeek> {

    public static DayOfWeek mon = new DayOfWeek("mon", 0);
    public static DayOfWeek tues = new DayOfWeek("tues", 1);
    public static DayOfWeek wed = new DayOfWeek("wed, 2");
    public static DayOfWeek thurs = new DayOfWeek("thurs", 3);
    public static DayOfWeek fri = new DayOfWeek("fri", 4);
    public static DayOfWeek sat = new DayOfWeek("sat", 5);
    public static DayOfWeek sun = new DayOfWeek("sun", 6);

    private DayOfWeek(String id, int ord) {
        super(id, ord);
    }

}
```

```
public class DayOfWeek
        extends Selector<DayOfWeek> {

    private final ReadAttribute<String> name;

    public static DayOfWeek mon = new DayOfWeek("Monday", "mon", 0);
    public static DayOfWeek tues = new DayOfWeek("Tuesday", "tues", 1);
    public static DayOfWeek wed = new DayOfWeek("Wednesday", "wed, 2");
    public static DayOfWeek thurs = new DayOfWeek("Thursday", "thurs", 3);
    public static DayOfWeek fri = new DayOfWeek("Friday", "fri", 4);
    public static DayOfWeek sat = new DayOfWeek("Saturday", "sat", 5);
    public static DayOfWeek sun = new DayOfWeek("Sunday", "sun", 6);

    private DayOfWeek(String name, String id, int ord)
    {
        super(id, ord);
        name = new SimpleReadAttribute(name);
    }

}
```

TODO: let each `case` override methods, like in Java.

2.3. Interfaces

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

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

```
InterfaceBody := "{" ( AbstractMethod | AbstractAttribute )* "}"
```

For example:

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

Which is equivalent to the following Java interface:

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

2.3.1. Interface inheritance

An interface may extend any number of other interfaces. For example:

```
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, and the above declaration is equivalent to the following Java declaration:

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

2.4. 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?
```

For example:

```
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) { ... }
```

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

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

A method may *only* refer to parameters in the first parameter list. It may not refer to parameters of other parameter lists.

A method implementation may be specified using either:

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

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

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

A method may be declared inside the body of another method or attribute, in which case it may refer to any immutable local of the containing scope. It may not refer to mutable locals.

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

```
@FormalParameterNames({"x", "y"})
public Integer add(Integer x, Integer y) { ... }
```

A Ceylon method invocation is equivalent to a Java method invocation. The semantics of method declarations are identical to Java, except that Ceylon methods may declare defaulted parameters.

2.4.1. Defaulted parameters

Methods with defaulted parameters may not be overloaded.

When a method with a defaulted parameter is called, and a value is not assigned to the defaulted parameter by the caller, the default value defined by the specifier is used.

This method:

```
public class Counter {

    package void init(Natural initialCount=0) {
        count:=initialCount;
    }

    ...

}
```

Is equivalent to three Java method declarations and an inner class:

```
public class Counter {

    void init() {
        init(0);
    }

    void init(Natural initialCount) {
        count=initialCount;
    }

    void init(CounterInitParameters namedParameters) {
        init( namedParameters.initialCount );
    }

    static class CounterInitParameters {
        private Natural initialCount=0;
        CounterInitParameters initialCount(Natural initialCount) {
            this.initialCount = initialCount;
            return this;
        }
    }

}
```

This named parameter call:

```
counter.init { initialCount=10; }
```

Is equivalent to this Java code:

```
counter.init ( new CounterInitParameters().initialCount(10) );
```

2.4.2. 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 may not specify an implementation:

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

Interface methods and abstract methods must be implemented by every non-`abstract` class that implements the interface or subclasses the abstract class.

2.5. Attributes

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

- Simple attribute declarations define state (very similar to a Java field).
- 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. If the attribute is not declared `optional`, it may not be overridden or implemented by an attribute declared `optional`.

For example:

```
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;
}
```

An attribute may be declared inside the body of another method or attribute, in which case it may refer to any immutable local of the containing scope. It may not refer to mutable locals.

An attribute declaration is equivalent to a Java method declaration together with a Java field declaration, both of type `lang.Attribute` or `lang.ReadAttribute`, both with the same name as the attribute.

The compiler is permitted to optimize private attributes to a simple Java field declaration or a local variable in a Java constructor. Private attributes may not be accessed via reflection.

TODO: Should we generate getters and setters, just for interop with Java?

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

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

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

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

2.5.1. Simple attributes and locals

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

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

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

The value of an immutable attribute is specified using `=`. Mutable attributes may be initialized using the assignment operator `:=`.

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

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

A local is really just a special case of a simple attribute declaration, but one that is optimized by the compiler.

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

The semantics of locals are identical to Java local variables.

For a simple attribute that is not a local, the Java field is initialized to an instance of `lang.SimpleAttribute` or `lang.SimpleReadAttribute`. For example:

```
package mutable String firstName;
```

is equivalent to this Java code:

```
private final Attribute<String> firstName = new SimpleAttribute<String>();
Attribute<String> firstName() { return firstName; }
```

While:

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

is equivalent to this Java code:

```
private final Attribute<Natural> count = new SimpleAttribute<Natural>(0);
private Attribute<Natural> count() { return count; }
```

And:

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

is equivalent to this Java code:

```
private final ReadAttribute<Integer> max = new SimpleReadAttribute<Integer>(99);
public ReadAttribute<Integer> max() { return max; }
```

2.5.2. 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.

For an attribute getter, the Java field is initialized to an instance of an anonymous inner subclass of `lang.Attribute` or `lang.ReadAttribute` that overrides the `get()` method with the content of the getter block. For example:

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

is equivalent to this Java code:

```
private final ReadAttribute<Float> total = new ReadAttribute<Float>() {
    @Override public Float get() { return items.get().totalPrice; }
};
public ReadAttribute<Float> total() { return total; }
```

2.5.3. Attribute setters

An attribute setter is declared as follows:

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

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

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

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

For an attribute with a setter, the Java field is initialized to an instance of an anonymous inner subclass of `lang.Attribute` that overrides the `set()` method with the content of the setter block. For example:

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

is equivalent to this Java code:

```
private final Attribute<String> name = new Attribute<String>() {
    @Override public String get() { return join(firstName, lastName); }
    @Override public void set(String name) { firstName = first(name); lastName = last(name); }
};
public Attribute<String> name() { return name; }
```

2.5.4. Interface attributes and abstract attributes

If there is no initializer or getter implementation, and the attribute is not declared `optional`, 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 may not specify an initalizer, getter or setter:

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

Interface attributes and abstract attributes must be implemented by every non-`abstract` class that implements the interface or subclasses the abstract class.

Interface attributes and abstract attributes may be specified `mutable`, in which case every subtype must also define the attribute to be mutable.

2.6. 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.

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

For example:

```
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;
```

2.7. 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 method must be a static method with exactly one parameter, or a non-static method with no parameters. An extension class must have exactly one initialization parameter. An extension method or attribute must have a non-`optional` type.

For example:

```
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;
    }

    ...
}
```

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

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.

The Ceylon compiler searches for an appropriate extension whenever a value of one type is assigned to a non-assignable type. If exactly one extension for the two types is found, the compiler inserts a call to the extension. For example, this Ceylon assignment:

```
import org.mydomain.myproject.Converters.personToUser;
...
Person person = ...;
User user = person;
```

Is equivalent to the following Java code:

```
Person person = ...;
User user = personToUser(person);
```

The Ceylon compiler also searches for an appropriate extension whenever a member is invoked that is not declared by the type. If exactly one extension that declares the member is found for the type, the compiler inserts a call to the extension. For example, this Ceylon method call:

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

Is equivalent to this Java code:

```
Collection<Integer> ints = ...;
Collection<Integer> result = new CollectionUtils(collection).nonZeroElements();
```

An extension is only available in a source file that explicitly `imports` the extension (except for the extensions defined in the package `lang`). A wildcard `.*`-style import may not be used to import an extension.

2.8. Declaration modifiers

The following annotations are compiler instructions:

- `public`, `module`, `package` determine the visibility of a declaration (by default, the declaration is visible only inside the same compilation unit).
- `abstract` specifies that a class cannot be instantiated.
- `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 a value may be null.
- `final` indicates that a class may not be extended, or that a method or attribute may not be overridden.
- `override` indicates that a method or attribute overrides a method or attribute defined by a supertype.
- `extension` specifies that a method is a converter, or that a class is a decorator.
- `once` indicates that a method is executed at most once, and the resulting value is cached.
- `deprecated` indicates that a method, attribute or type is deprecated.
- `volatile` indicates a volatile simple attribute.

TODO: We can minimize backtracking in the parser by making all these "annotations" be keywords. It lets the parser recognize a member declaration a little bit more easily. But on the other hand it's a new special kind of thing.

TODO: should there be an `annotation` modifier for static methods which can be used as annotations?

TODO: does Ceylon support static attributes?

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 following annotations are instructions to the documentation compiler:

- `doc` specifies the documentation for 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 values of the `doc`, `throws` and `by` annotations are parsed by the documentation compiler as Seam Text, a simple ANTLR-based wiki text format.

The following annotations are important to the Ceylon SDK.

- `id` specifies that an attribute should be tested by the `equals()` method, and included in the `hash`.
- `transient` specifies that an attribute is not included in the serialized form of the object.
- `read` and `write` indicate methods or attributes that are protected from multithreaded access using a reentrant read/write lock with deadlock detection.

Chapter 3. 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.

3.1. 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 := "{" DeclarationOrStatement* DirectiveStatement? "}"
```

```
DeclarationOrStatement := Declaration | Statement
```

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
```

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

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

3.1.1. 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 ) ";"
```

```
Invocation := MethodInvocation | StaticMethodInvocation | Instantiation
```

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.

3.1.2. 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 ";"?
```

For example:

```
throw Exception()
```

```
return x+y;
return "Hello"
```

```
break true
```

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, and
- the `throw` directive—to raise an exception.

```
Directive := "return" Expression? | "throw" Expression? | "break" Expression?
```

The `return` directive may not be used outside the body of an attribute getter or method. A return value must be specified in a non-`void` method or attribute getter. A return value may not be specified in a `void` method.

The `break` directive may not be used outside the body of a loop. If the loop has a `fail` block, the `break` directive must specify a boolean value. A value of `false` specifies that the `fail` block should be executed. If the loop has no `fail` block, the `break` directive may not specify a value.

A `throw` directive which specifies no exception is equivalent to `throw Exception()`.

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.

3.1.3. Specification statements

A specification statement may specify the value of an immutable 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 an immutable local or attribute (using `=`). A specification is not an expression.

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;
}
```

3.1.4. Nested declarations

Blocks may contain declarations, which are, by default, only visible inside the block:

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

3.2. Control structures

Control of execution flow may be achieved using control directives and control structures. Control structures include conditionals, loops, and exception management.

Control structures are not considered to be expressions in Ceylon.

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
```

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

```
Variable := Type MemberName
```

Some control structures expect conditions:

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

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

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

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 must be an expression of type `Boolean`. The condition is satisfied if the expression evaluates to `true` at runtime.
- If the `is` modifier appears, the condition must be an expression or local specifier of type `Object`. The condition is satisfied if the expression or specifier evaluates to an instance of the specified type at runtime.
- If the `exists` modifier appears, the condition must be an expression or local specifier of type `optional Object`. The condition is satisfied if the expression or specifier evaluates to a non-null value at runtime.
- If the `nonempty` modifier appears, the condition must be an expression or local specifier of type `optional Container`. The condition is satisfied if the expression or specifier evaluates to a non-null value at runtime, and if the resulting `Container` is non-empty.

For `exists` or `nonempty` conditions:

- if the condition is a local variable specifier, the local variable may be declared without the `optional` annotation, even though the specifier expression is of type `optional`, or
- if the condition is a local variable, the local will be treated by the compiler as having non-null type inside the block that follows immediately, relaxing the usual compile-time restrictions upon `optional` types.

For `is` conditions:

- if the condition is a local variable specifier, the local variable may be declared with the specified type, even though the specifier expression is not of that type, or
- if the condition is a local variable, the local will be treated by the compiler as having the specified type inside the block that follows immediately.

TODO: Should we support, like Java, single-statement control structure bodies without the braces.

3.2.1. `if/else`

The `if/else` conditional has the following form:

```
IfElse :=
"if" "(" Condition ")" Block
("else" "if" "(" Condition ")" Block)*
("else" Block)?
```

If the condition is satisfied, the first block is executed. Otherwise, the second block is executed, if it is defined.

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;
}
```

3.2.2. `switch/case/else`

The `switch/case/else` conditional has the following form:

```
SwitchCaseElse := "switch" "(" Expression ")" (Cases | "{" Cases "}")
```

```
Cases :=
("case" "(" Case ")" Block)+
("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`.

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

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

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

If no `else` block is defined, the switch expression must be of type `Selector`, and all enumerated `cases` of the class must be explicitly listed.

If the switch expression is of type `Selector`, and all enumerated `cases` of the class are explicitly listed, no `else` block may be specified.

When the construct is executed, the switch expression value is tested against the case values using `Case.test()`, and the case block for the first case value that tests true is executed. If no case value tests true, and an `else` block is defined, the `else` block is executed.

For an `is` type 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.

TODO: support `catch`-style syntax instead of `case (is ...)`?

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");
    }
}
```

3.2.3. `for/fail`

The `for/fail` loop has the following form:

```
ForFail :=
"for" "(" ForIterator ")" Block
("fail" Block)?
```

An iteration variable declaration must specify an iterated expression that contains the range of values to be iterated.

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

Each iterated expression must be of type `Iterable` or `Iterator`. If two iteration variables are defined, it must be of type `Iterable<Entry>` or `Iterator<Entry>`.

The body of the loop is executed once for each iterated element.

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 is executed, if it is defined.

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");
}
```

3.2.4. `do/while`

The `do/while` loop has the form:

```
DoWhile :=
( "do" ("(" DoIterator ")")? Block? )?
"while" "(" Condition ")" (";" | Block)
```

The loop may declare an iterator variable, which may be mutable.

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

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?

3.2.5. `try/catch/finally`

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

```
TryCatchFinally :=
"try" ( "(" Resource ("," Resource)* ")" )? Block
("catch" "(" Variable ")" Block)*
("finally" Block)?
```

When an exception occurs in the try block, the first matching catch block is executed, if any. The finally block is always executed.

The type of each catch local must extend `lang.Exception`.

Each resource expression must be of type `Usable`.

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

When the construct is executed, `begin()` is called upon the resource, the try block is executed, and then `end()` is called upon the resource, with the thrown exception, if any.

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
}
```

Chapter 4. Expressions

Ceylon expressions are significantly more powerful than Java, allowing a more declarative style of programming.

Expressions are formed from:

- literal values,
- invocation of methods and instantiation of classes and enumerations,
- evaluation and assignment of attributes, and
- operators.

4.1. Literals

Ceylon supports literal values of the following types:

- `Natural` and `Float`,
- `String` and `Quoted`,
- `Type`, `Attribute` and `Method`.

```
Literal := NaturalLiteral | FloatLiteral | StringLiteral | QuotedLiteral | TypeLiteral | MemberLiteral
```

Ceylon does not need a special syntax for `Boolean` literal values, since `Boolean` is just a `Selector` with the enumerated instances `true` and `false`.

TODO: Should we have a literal syntax for version numbers, for use in the module architecture, for example: `@1.2.3BETA`?

4.1.1. Natural number literals

A `Natural` literal has this form:

```
NaturalLiteral = Digit+
```

For example:

```
Natural m = n + 10;
```

Negative integer values can be produced using the unary `-` operator:

```
Integer i = -1;
```

4.1.2. Floating point number literals

A `Float` literal has this form:

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

For example:

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

Equivalent to this Java code:

```
public static final Float pi = new lang.Float(3.14159f);
```

4.1.3. String literals

A `String` literal has this form:

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

For example:

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

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

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

The first example is equivalent to this Java code:

```
person.name().set( new lang.String("Gavin") );
```

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?

4.1.4. 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 `Quoted` literal has this form:

```
QuotedLiteral := "'" Character* "'"
```

For example:

```
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';
```

```
Character x = 'x';
```

```
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 ...; }
```

4.1.5. Type and member literals

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 literal syntax:

```
TypeLiteral := HASH Type
```

```
MemberLiteral := HASH (Type ".")? MemberName
```

For example:

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

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

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

4.2. `this`, `super`, `null` and `subtype`

```
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. Any invocation of this reference is processed by the method 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.

The keyword `none` refers to a special value that is assignable to all types that extend `Enumeration`. This value has no elements.

4.3. Invocations

Methods and classes are *invokable*. Invocation of a class is called *instantiation*.

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

```
Arguments := PositionalArguments | NamedArguments
```

Required parameters must be specified. Defaulted parameters and varargs may also be specified.

4.3.1. Positional arguments

When parameter values are listed, required parameters are specified first, in the order in which they were declared, followed by defaulted parameters, in the order they were declared. If there are any remaining defaulted parameters, they will be assigned their default values. On the other hand, if any parameter values are unassigned, they will be treated as varargs.

```
PositionalArguments := "(" ( Expression ("," Expression)* )? ")"
```

For example:

```
(getProduct(id), 1)
```

4.3.2. Named arguments

When parameter values are named, required and defaulted parameter values are specified by name. Vararg parameter values are specified by listing them.

```
NamedArguments := "{" NamedArgument* VarargArguments? "}"
```

A named argument either:

- specifies its value using =, and is terminated by a semicolon, or
- only for functional parameters, specifies a formal parameter list and a block of code (an inline method declaration).

```
NamedArgument := ParameterName (Specifier ";" | FormalParams Body)
```

For example:

```
{
    product = getProduct(id);
    quantity = 1;
}
```

```
{
    label = "Say Hello";
    onClick() {
        say("Hello");
    }
}
```

```
{
    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 an enumeration instantiation to be omitted in this case?

4.3.3. 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} )
```

4.3.4. Method invocation

Method invocations follow this schema:

```
MethodInvocation := MemberReference TypeArguments? Arguments
```

```
MemberReference := (Expression ".")? MemberName
```

For example:

```
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 }
```

The value of a method invocation is the return value of the method. The parameter values are passed to the formal parameters of the method.

Methods may not be invoked using the `.` operator on an expression of type `optional`. They may be invoked using `?.`.

4.3.5. Static method invocation

Static method invocations follow this schema:

```
StaticMethodInvocation := StaticMemberReference TypeArguments? Arguments
```

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

For example:

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

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

The value of a static method invocation is the return value of the static method. The parameter values are passed to the formal parameters of the method.

4.3.6. Class instantiation

Classes may be instantiated according to the following schema:

```
Instantiation := RegularType Arguments
```

For example:

```
Map<String, Person>(entries)
```

```
Point { x=1.1; y=-2.3; }
```

```
ArrayList<String> { capacity=10; "gavin", "max", "emmanuel", "steve", "christian" }
```

```
Panel {
    label = "Hello";
    Input i = Input(),
    Button {
        label = "Hello";
        action() {
            log.info(i.value);
        }
    }
}
```

The value of a class instantiation is a new instance of the class. The parameter values are passed to the initialization parameters of the class. If the class has no initialization parameters, they are assigned directly to attributes of the class (in this case, named parameters must be used).

4.3.7. Enumerations

Enumerations may be instantiated according to the following simplified syntax:

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

This is a shortcut for instantiation of the `Enumeration` class, where there is no need to explicitly specify the type.

For example:

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

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

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

There are no true literals for lists, sets or maps. However, the `..` and `->` operators, together with the convenient enumeration constructor syntax, and some built-in extensions help us achieve the desired effect.

```
List<String> languages = { "Java", "Ceylon", "Smalltalk" };
```

```
List<Natural> numbers = 1..10;
```

Enumerations 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.

4.3.8. Attribute evaluation and assignment

Attribute evaluation follows this schema:

```
AttributeGet := MemberReference
```

This attribute evaluation:

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

is equivalent to the following Java code:

```
String name = person.name().get();
```

Attribute assignment follows the following schema:

```
AttributeSet := MemberReference AssignmentOperator Expression
```

This attribute assignment:

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

is equivalent to the following Java code:

```
person.name().set("Gavin");
```

If getter code is specified, and `assign` is not specified, the attribute is not settable, and any attempt to assign to the attribute will result in a compiler error.

Attributes may not be accessed using the `.` operator when the expression is of type `optional`. They may be accessed using `?.`.

Attributes may not be assigned when the expression of type `optional`.

4.4. Method references

A method reference has the form:

```
MethodReference :=  MemberReference | StaticMemberReference
```

where the member name is the name of a method.

Method references may appear as an argument to a functional parameter, either as a positional argument, or on the RHS of an = specifier in a named parameter invocation. The signature of the functonal parameter must match the signature of the method. For example:

```
//a method which accepts a method reference
void sort(List<String> list, Comparison by(String x, String y)) { ... }
```

```
Comparison reverseAlpha(String x, String y) { return y<=>x; } //a local method declaration
sort(names, reverseAlpha); //pass a reference to the method to another method
```

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

```
//a method that returns a method reference
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;
    }
}
```

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

Finally, a method reference may be used to define a method using `=`. The two methods must have exactly the same signature.

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

TODO: attribute references `ref person.name`, which may be specified to attributes or parameters declared `reference` or returned by methods declared `reference`.

4.5. Assignable expressions

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

The following expressions are assignable:

- a local declared `mutable`, for example `count`,
- any attribute expression where the underlying attribute has a setter or is a simple attribute declared `mutable`, for example `person.name`, and
- element expressions for the type `OpenCorrespondence`, for example `order.lineItems[0]`

When an assignment expression is executed, the value of the local or attribute is set to the new value, or the `define()` method of `OpenCorrespondence` is called.

Thus, the following statement:

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

Is equivalent to the Java:

```
order.lineItems.define( 0, new LineItem(prod, 1) );
```

4.6. 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 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 = ArrayList<Person>()^.append(person1)^.append(person2)*.name;
```

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

4.6.1. List of operators

The following tables define the semantics of the Ceylon operators:

4.6.1.1. 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 4.1.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Assignment						

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
`:=`	`lhs := rhs`	assign		`X` or `optional X`	`X` or `optional X`	`X` or `optional X`
Member invocation						
`.`	`lhs.member`	invoke		`X` or type `X`	Member of `X`	Member type
`^.`	`lhs^.member`	chain invoke		`X`	Member of `X`	`X`
`?.`	`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 `X`	`List` of member type
Render						
`$`	`$rhs`	render	`if (exists rhs) rhs.string else ""`		`optional Object`	`String`
Compound invocation assignment						
`.=`	`lhs.=member`	apply	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

4.6.1.2. Equality and comparison operators

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

Table 4.2.

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`

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

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

4.6.1.3. Logical operators

These are the usual logical operations for boolean values.

Table 4.3.

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 lhs:=rhs`	`Boolean`	`Boolean`	`Boolean`
`&&=`	`lhs &&= rhs`	conditional and	`if (lhs) lhs:=rhs else false`	`Boolean`	`Boolean`	`Boolean`

4.6.1.4. Operators for handling null values

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

Table 4.4.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Existence						

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
`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`
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 lhs:=rhs`	`optional T`	`T` or `optional T`	`T` or `optional T`

4.6.1.5. List and map operators

These operators provide a simplified syntax for accessing values of a `Correspondence`, and for joining `Lists`.

Table 4.5.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Concatenation						
`+`	`lhs + rhs`	join	`lhs.join(rhs)`	`List<X>`	`List<X>`	`List<X>`
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]`	lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`List<X>`	`List<Y>`
`[]`	`lhs[indices]`	lookup	`lhs.values(index)`	`Correspondence<X,Y>`	`Set<X>`	`Set<Y>`
Subranges						
`[..]`	`lhs[x..y]`	subrange	`lhs.range(x,y)`	`List<X>`	Two `Natural` values	`List<X>`
`[...]`	`lhs[x...]`	upper range	`lhs.range(x)`	`List<X>`	`Natural`	`List<X>`

TODO: Should we overload `*` for lists, 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?

4.6.1.6. Operators for constructing objects

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

Table 4.6.

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?

4.6.1.7. Arithmetic operators

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

Table 4.7.

Op	Example	Name	Equivalent	LHS type	RHS type	Return type
Increment, decrement						
`++`	`++rhs`	successor	`rhs := rhs.successor`		`Ordinal<T>`	`T`
`--`	`--rhs`	predecessor	`rhs := rhs.predecessor`		`Ordinal<T>`	`T`
`++`	`lhs++`	increment	`(lhs := lhs.successor).predecessor`	`Ordinal<T>`		`T`
`--`	`lhs--`	decrement	`(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`
Numeric assignment						
`+=`	`lhs += rhs`	add	`lhs := lhs.plus(rhs)`	`Numeric<N,I>`	`N`	`N`
`-=`	`lhs -= rhs`	subtract	`lhs := lhs.minus(rhs)`	`Numeric<N,I>`	`N`	`I`
`*=`	`lhs *= rhs`	multiply	`lhs := lhs.times(rhs)`	`Numeric<N,I>`	`N`	`N`
`/=`	`lhs /= rhs`	divide	`lhs := lhs.divided(rhs)`	`Numeric<N,I>`	`N`	`N`
`%=`	`lhs %= rhs`	remainder	`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`.

4.6.1.8. 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 4.8.

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	`lhs := lhs.or(rhs)`	`Bits<X>`	`Bits<X>`	`X`
`&=`	`lhs &= rhs`	and	`lhs := lhs.and(rhs)`	`Bits<X>`	`Bits<X>`	`X`
`^=`	`lhs ^= rhs`	exclusive or	`lhs := lhs.xor(rhs)`	`Bits<X>`	`Bits<X>`	`X`

4.6.2. Operator precedence and associativity

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

Table 4.9.

Operations	Operators	Type	Associativity
Member invocation and lookup, subrange, postfix increment, postfix decrement:	`.`, `^.`, `*.`, `?.`, `(,,)`, `{;;;}`, `[]`, `[..]`, `[...]`, `++`, `--`	Binary / ternary / N-ary / unary postfix	Left
Prefix increment, prefix decrement, negation, render, bitwise complement:	`++`, `--`, `-`, `$`, `~`	Unary prefix	Right
Exponentiation:	`**`	Binary	Right
Multiplication, division, remainder, bitwise and:	`*`, `/`, `%`, `&`	Binary	Left

Operations	Operators	Type	Associativity
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 `**`.

4.7. Smalltalk-style method invocation

Ceylon provides a special method invocation protocol inspired by Smalltalk for specifying arguments to functional parameters of a method. This special invocation protocol may be used to imitate the syntax of built-in control structures. For example:

```
sort(amounts) by (Float x, Float y) { return x<=>y };
```

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

```
x>10
isTrue {
    log.debug("big");
    big(x);
}
isFalse {
    log.debug("little");
    little(x);
};
```

In a Smalltalk-style method invocation, an argument to a functional parameter may be specified with certain punctuation eliminated:

- if the functional parameter declares no parameters, the empty parentheses may be eliminated, and
- if the body of the method implementation consists of a single `return` directive, the braces and `return` keyword may be eliminated.

These arguments are called *functional arguments* of a positional parameter invocation.

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

```
FunctionalArgument := ParameterName FunctionalBody
```

For example:

```
elements (Person p) p.age>18
```

```
isTrue x+1
```

```
each { count+=1; }
```

Functional arguments are listed without any seperating punctuation:

```
isTrue x+1 isFalse 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.

The method invocation protocol begins with either:

- a normal method invocation with a period before the method name and an ordered list of arguments surrounded be parentheses, or
- the receiving expression, followed by the the method name, without the period or parentheses, followed immediately by the first block argument.

```
MethodInvocationWithFunctionalArguments :=
( MemberReference TypeArguments? PositionalArguments | Expression MemberName FunctionalBody )
FunctionalArgument*
```

Some additional examples:

```
namedValues each (String name->Object value) {
    log.info("${name} ${value}");
};
```

```
Set<People> adults = people elements (Person p) p.age>18;
```

```
int index = lineItems firstIndex (LineItem li) !li.product.available;
```

```
people sort (Person p, Person q) p.name<=>q.name;
```

```
String label = x>10 isTrue "big" isFalse "little";
```

```
optional specialPerson = search (people) findFirst (Person p) p.special orIfNoneFound null;
```

TODO: there is a small problem here. In `x.foo(y) bar z`, is `bar` the name of a method or of a parameter of `foo()`? That's not such a big deal for the compiler, but for the human reader I guess it's going to get confusing occasionally. So should we play it safe and require the form `x.foo(y).bar() baz z` in the case that `bar` is the method name?

TODO: On the other hand, if we don't want to play things safe, should we allow this invocation protocol for methods with no parameters, or with arguments which are not functional (I have checked that this can be parsed), for example:

```
Integer x = y integer;
```

```
log info "Hello World";
```

```
log.info(", " join org.employees*.name);
```

4.7.1. Iteration

A specialized invocation syntax is provided for 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:

```
MemberReference "(" ForIterator ")" BlockArguments
```

And each functional argument for a parameter annotated `coordinated` with a single parameter of type `X` need not declare its formal parameter. Instead, its parameter is declared by the iterator.

For example, for the following method declaration:

```
public static
List<Y> from<X,Y>(iterated Iterable<X> elements,
                  coordinated Boolean having(X x),
                  coordinated Y select(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}";
```

4.7.2. Local definition

A specialized invocation syntax is also provided for methods which define a local. 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:

```
MemberReference "(" Variable Specifier ")" BlockArguments
```

And each functional argument for a parameter annotated `coordinated` with a single parameter of type `X` need not declare its formal parameter. Instead, its parameter is declared by the specifier.

For example, for the following method declaration:

```
public static
Y ifExists<X,Y>(specified optional X value,
                coordinated Y then(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,
             coordinated Y seek(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);
```

Chapter 5. 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*.

5.1. 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), `^.` (chain 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?

5.2. 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 final 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)
    }
}
```

5.3. Iterable objects and iterators

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> {

    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;

}
```

5.4. 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> { ... }
```

5.5. 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);

}
```

5.6. Category, Correspondence and Container

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);

}
```

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 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 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
    }

}
```

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);
         }
     }

}
```

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;

}
```

5.7. 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)
    }

}
```

5.8. Collections

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

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

    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));

}
```

5.8.1. 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> {}
```

5.8.2. Lists

Lists implement the following interface, and support the binary operators `+` (join), `[i...]` (upper range) and the ternary operator `[i..j]` (subrange) in addition to operators inherited from `Collection` and `Correspondence`:

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

    doc "The first element of the list."
    throws #EmptyException
           "if the list is empty"
    public X first;

    doc "The last element of the list."
    throws #EmptyException
           "if the list is empty"
    public X last;

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

    doc "The first element of the list, or null if the
         list is empty."
    public optional X firstOrNull;

    doc "The last element of the list, or null if the
         list is empty."
    public optional X lastOrNull;

    doc "The index of the last element of the list."
    throws #EmptyException
           "if the list is empty"
    public Natural lastIndex;

    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 "The ternary range operator x[from..to], along
         with the binary upper range x[from...] operator.
         The returned list does not reflect changes to
         the original list."
    public List<X> range(Natural from, Natural to=size);

    doc "The binary join operator x + y.
         The returned list does not reflect changes
         to the original lists."
    public List<T> join(List<T> elements) where T<=X;

    doc "A sublist from the first given index (inclusive)
         to the second given index (exclusive). The size
         of the returned sublist is the difference between
         the two indexes. The returned list does reflect
         changes to the original list."
    public List<X> sublist(Natural from, Natural to=size);

    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 ( 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>, OpenCorrespondence<Natural, X> {

    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 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 "Remove elements beginning with the first given index
         which have an index less than the second given index,
         decrementing the indexes of all elements beginning
         with the second given index by the difference between
         the two indexes."
    public void delete(Natural from, Natural to=size);

    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;
    }

}
```

5.8.3. 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;

}
```

5.8.4. 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;

}
```

5.8.5. 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> {
            ...
        }
    }

}
```

5.9. 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."
    public subtype successor;

    doc "The unary "--" operator. The predecessor of this instance."
    public subtype predecessor;

}
```

5.10. Ranges and enumerations

Ranges and enumerations both implement `List`, 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 List<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 List<X> by(Natural stepSize);

}
```

Enumerations represent an explicit list of values and may be constructed using a simplified syntax:

```
public class Enumeration<out X>(X... values)
        satisfies List<X>, Case<Object> {
    ...

    public static extension
    OpenMap<U, V> toOpenMap<U, V>(Enumeration<Entry<U,V>> enum) {
        return HashMap(enum)
    }

    public extension
    OpenSet<X> toOpenSet<X>() {
        return elements.copy()
    }

    public extension
    OpenList<X> toOpenList<X>() {
        return copy()
    }

    public static extension
    Enumeration<T> singleton<T>(T object) {
        return Enumeration(object)
    }

}
```

5.11. 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 `List`, therefore they support the join, subrange, contains and lookup operators, among others.

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

    doc "Collapse substrings of the given characters into
         single space characters."
    public String normalize(Character... whitespace = " \n\l\r\t") { return ... }

    doc "Join the given strings, using this string as
         a separator."
    public String join(String... strings) { return ... }

}
```

What encoding is the default for Ceylon strings? Are there performance advantages to going with UTF-16 like Java? Can we easily abstract this stuff? Does the JVM do all kinds of optimizations for `java.lang.String`?

5.12. 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?

5.13. Bit strings

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

```
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;

}
```

5.14. 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 final 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 final 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 final 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 final 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 final 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 ... }

    }

}
```

5.15. 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;
}
```

5.16. 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() && !evaluate() ) {
            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,
                                           coordinated Y then(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,
                                  coordinated Y then(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,
                                   coordinated Boolean having(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,
                                    coordinated Boolean every(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,
                                    coordinated Boolean some(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,
                                      coordinated Boolean having(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,
                             coordinated Boolean having(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,
                                  coordinated Boolean having(X x),
                                  optional coordinated Comparison by(X x, X y) = 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,
                                    coordinated Y select(X x),
                                    optional coordinated Comparable by(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,
                                    coordinated Boolean having(X x),
                                    coordinated Y select(X x),
                                    optional coordinated Comparable by(Y x, Y y) = 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,
                                        coordinated V to(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,
                                      coordinated V from(U key)) {
        return map(values, (V value) value->from(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,
                                      coordinated Entry<U,V> of(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,
                               coordinated Y seek(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,
                                 coordinated Y seek(X x),
                                 Y except(E e))
                    where X >= Usable {
        try (resource) {
            return seek(resource)
        }
        catch (E e) {
            return except(e)
        }
    }
}
```

5.17. 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.