Chapter 1. Introduction in Ceylon

1.1. Literal values

Ceylon supports literal values of the following types:

- `Integer`
- `Float`
- `Character`
- `String`

The types `Integer`, `Float`, `Character`, and `String` are defined in the module ceylon.language.

1.1.1. Integer number literals

An integer literal is an expression of type `Integer`, representing a numeric integer.

```
Integer five = 5;
```

```
Integer mask = $1111_0000;
```

```
Integer white = #FFFF;
```

However they can be a bit more sophisticated. The digits of a numeric literal may be grouped using underscores. If the digits are grouped, then groups must contain exactly three digits.

```
Integer twoMillionAndOne = 2_000_001;
```

A hexadecimal integer is written using a prefix #. Digits may be grouped into groups of two or four digits.

```
Integer white = #FF_FF_FF;
```

A binary integer is written with a prefix $. Digits may be grouped into groups of four digits.

```
Integer sixtyNine = $0100_0101;
```

1.1.2. Floating point number literals

A floating point literal is an expression of type `Float`, a floating-point representation of a numeric value.

```
shared Float pi = 3.14159;
```

```
shared Float oneHundredth = 0.01
```

```
shared Float pi = 3.141_592_654;
```

A very large or small numeric literal may be qualified by one of the standard SI unit prefixes: m, u, n, p, f, k, M, G, T, P.

```
Float red = 390.0n;                 // n (nano) means E-9
Float galaxyDiameter = 900.0P;      // P (peta) means E+15
Float hydrogenRadius = 25.0p;       // p (pico) means E-12
Float usGovDebt = 14.33T;           // T (tera) means E+12
Float brainCellSize = 4.0u;         // u (micro) means E-6
Integer deathsUnderCommunism = 94M;  // M (mega) means E+6
```

1.1.3. Character literals

A single character literal is an expression of type `Character`, representing a single character.

```
if (exists ch=string[i], ch == '+') { ... }
```

Ceylon strings are composed of Characters—indeed, a `String` is a `List` of `Characters`. A character literal is written between single quotes.

```
Character[] latinLetters = concatenate('a'..'z', 'A'..'Z');
           Character newline = '\n';
           Character pi = '\{#0001D452}';
```

An instance of `Character` represents a 32-bit Unicode character, not a Java-style UTF-16 char. However, under the covers, Ceylon strings are implemented using a Java `char[]` array (in fact, they are implemented using a Java string). So some operations on Ceylon strings are much slower than you might expect, since they must take four-byte characters into account. This includes size and `item()`. We think it's much better that these operations be slow, like in Ceylon, than that they sometimes give the wrong answer, like in Java. And remember, it's never correct to iterate a list using size and `item()` in Ceylon! To avoid the cost of calling `size()`, try to use the more efficient `empty`, `longerThan()` and `shorterThan()` when the string might be very long.

1.1.4. Character string literals

A character string literal or verbatim string is an expression of type `String`, representing a sequence of characters. A *string literal* is text enclosed in double-quotes:

```
String name = "Gavin King";
print(name);
```

```
print( "Hello, World!");
```

```
String verbatim = """A verbatim string can have \ or a " in it.""";
```

String literals in Ceylon may span multiple lines. Try this:

```
print("Hello,
       World!");
```

The output is:

```
Hello,
World!
```

1.2. Basic

1.2.1. Identifiers and keywords

Identifiers may contain letters, digits and underscores.

The following examples are legal identifiers:

```
Person
```

```
name
```

```
personName
```

```
_id
```

```
x2
```

```
\I_id
```

```
\Iobject
```

```
\iObject
```

```
\iclass
```

The prefix `\I` or `\i` is not considered part of the identifier name. Therefore, `\iperson` is just an initial lowercase identifier named `person` and `\Iperson` is an initial *uppercase* identifier named `person`.

1.2.2. Comments

There are two kinds of comments:

- a *multiline comment* begins with `/*` and extends until `*/`, and
- an *end-of-line comment* begins with `//` or `#!` and extends until the next line terminating character sequence.

Both kinds of comments can be nested.

The following examples are legal comments:

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

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

```
#!/usr/bin/ceylon
```

1.2.3. Escape sequence and string concatination

Inside a string literal, you can use the escape sequences `\n`, `\t`, `\\`, `\"` and friends that you're used to from other C-like languages.

```
print("\"Hello!\", said the program.");
```

```
print("Hello, this is Ceylon ``language.version``
       running on ``runtime.name`` ``runtime.version``!\n
       You ran me at ``system.milliseconds`` ms,
       with ``process.arguments.size`` arguments.");
```

Notice how our message contains interpolated expressions, delimited using "doublebacks", that is, two backticks. This is called a string template. On my machine, this program results in the following output:

```
Hello, this is Ceylon 1.0.0
running on jvm 1.7!

You ran me at 1362763185067 ms,
with 0 arguments.
```

The `+` operator you're probably used to is an alternative way to concatenate strings, and more flexible in many cases:

```
print("Hello, this is Ceylon " + language.version +
            " running on " + runtime.name + " " + runtime.version + "!\n" +
            "You ran me at " + system.milliseconds.string +
            " ms, with " + process.arguments.size.string +
            " arguments.");
```

Note that when we use `+` to concatenate strings, we have to explicitly invoke the string attribute to convert numeric expressions to strings. The `+` operator does not automatically convert its operands to strings, so the following does not compile:

```
print("Hello, this is Ceylon " + language.version +
            " running on " + runtime.name + " " + runtime.version + "!\n" +
            "You ran me at " + system.milliseconds +  //compile error!
            " ms, with " + process.arguments.size +    //compile error!
            " arguments.");
```

1.2.4. Optional types

Local variables, parameters, and attributes that may contain `null` values must be explicitly declared as being of optional type (the `T?` syntax). There's simply no way to assign `null` to a local that isn't of optional type. The compiler won't let you. This is an error:

```
String name = null; //compile error: null is not an instance of String
```

Nor will the Ceylon compiler let you do anything dangerous with a value of type `T?` - that is, anything that could cause a `NullPointerException` in Java - without first checking that the value is not null using `if (exists ... )`. The following is also an error:

```
String? name = process.arguments.first;
print("Hello " + name + "!"); //compile error: name is not Summable
```

In fact, it's not even possible to use the equality operator == with an expression of optional type. We can't write:

```
String? name = process.arguments.first;
if (name==null) { ... } //compile error: name is not Object
```

like we can in Java. This helps avoid the undesirable behavior of == in Java where `x==y` evaluates to true if x and y both evaluate to null.

In a language with static typing, we're always wanting to know what the type of something is. So what's the type of null? That's easy to answer: `null` is a `Null`. Yes, that's right: the value null isn't a primitive value in Ceylon, it's just a perfectly ordinary instance of the perfectly ordinary class `Null`, at least from the point of view of Ceylon's type system.

And the syntax String? is just an abbreviation for the union type `Null|String`. That's why we can't call operations of `String` on a `String?`. It's simply a different type!

The `if (exists ...)` construct narrowed the type of name inside the if block, allowing us to treat name as a `String` there. (As an aside, if you're concerned about performance, it's worth mentioning that the Ceylon compiler does some special magic to transform this value to a virtual machine-level null, all under the covers.)

1.2.5. Operators for handling null values

There are a couple of operators that will make your life easier when dealing with `null` values. The first is else:

```
String greeting = "Hello, " + (name else "World");
```

The else operator produces: its first operand if the first operand `is not null`, or its second operand `otherwise`. It's a more convenient way to handle `null` values in simple cases. You can chain multiple elses:

```
String name = firstName else userId else "Guest";
```

There's also an operator for producing a `null` value:

```
String? name = !arg.trimmed.empty then arg;
```

The then operator produces its second operand if its first operand evaluates to `true`, or `null` otherwise. You can chain an else after a then to reproduce the behavior of C's ternary ?: operator:

```
String name = !arg.trimmed.empty then arg else "World";
```

Finally, the `?.` operator lets us call operations on optional types:

```
Integer length = name?.size else 0;
```

If name is `null`, `name?.size` evaluates to `null`. Otherwise, the size attribute of `String` is evaluated. If we need to squeeze a whole chain of thens/elses into a single expression, we can use the "poorman's switch" idiom:

```
String sign = (int>1P then "enormous")
                else (int < 0 then "negative")
                    else (int > 0 then "positive")
                    else "zero";
```

Using else, we can finally simplify our original example to something reasonable:

```
print("Hello, ``process.arguments.first else "World"``!");
```

Yes, after all that, it's a one-liner ;-)

1.2.6. Numbers

Unfortunately, not every program is as simple and elegant as `"hello world"`. In business or scientific computing, we often encounter programs that do fiendishly complicated stuff with numbers. Ceylon doesn't have any primitive types, so numeric values are usually represented by the classes `Integer` and `Float`, which we'll come back to later in the tour. `Float` literals are written with a decimal point, and `Integer` literals without:

```
Integer one = 1;
Float zero = 0.0;
```

Even though they're classes, you can use all the usual numeric literals and operators with them. For example, the following function efficiently determines if an `Integer` represents a prime number:

```
"Determine if `n` is a prime number."
throws (`class AssertionError`, "if `n<2`")
Boolean prime(Integer n) {
   "`n` must be greater than 1"
   assert (n>1);
   if (n<=3) {
       return true;
   }
   else if (n%2==0 || n%3==0) {
       return false;
   }
   else if (n<25) {
       return true;
   }
   else {
       for (b in 1..((n.float^0.5+1)/6).integer) {
           if (n%(6*b-1)==0 || n%(6*b+1)==0) {
               return false;
               }
       }
       else {
           return true;
       }
   }
}
```

Try it, by running the following function:

```
"Print a list of all two-digit prime numbers."
void findPrimes()=> printAll { for (i in 2..99) if (prime(i)) i };
```

Heh, this was just a little teaser to keep you interested.

1.2.7. Numeric widening

Ceylon doesn't have implicit type conversions, not even built-in conversions for numeric types. Assignment does not automatically widen (or narrow) numeric values. Instead, we usually need to call one of the operations (well, attributes, actually) defined by the interface `Number`.

```
Float zero = 0.float; // explicitly widen from Integer
```

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

```
Float diagonal = (length^2.0+width^2.0)^0.5;
```

Of course, if you want to use the increment `++` operator, decrement `--` operator, or one of the compound assignment operators such as `+=`, you'll have to declare the value variable. Since it's quite noisy to explicitly perform numeric widening in numeric expressions, the numeric operators do automatically widen their operands, so we could write the expression above

like this:

```
Float diagonal = (length^2+width^2)^(1.0/2);
```

Since ceylon.language only has two numeric types the only automatic widening conversion is from Integer to `Float`. This is the one and only thing approaching an implicit type conversion in the whole language.

1.2.8. Operator polymorphism

Ceylon discourages the creation of intriguing executable ASCII art. Therefore, true operator overloading is not supported by the language. Instead, almost every operator (every one except the primitive `.`, `()`, `is`, `=`, `===`, and `of` operators) is considered a shortcut way of writing some more complex expression involving other operators and ordinary function calls. For example, the `<` operator is defined in terms of the interface `Comparable`, which has a method named `compare()`. The operator expression

```
x<y
```

means, by definition,

```
x.compare(y) === smaller
```

The equality operator `==` is defined in terms of the class `Object`, which has a method named `equals()`. So

```
x==y
```

means, by definition,

```
x.equals(y)
```

Therefore, it's easy to customize operators like `<` and `==` with specific behavior for our own classes, just by implementing or refining methods like `compare()` and `equals()`. Thus, we say that operators are polymorphic in Ceylon. Apart from `Comparable` and `Object`, which provide the underlying definition of comparison and equality operators, the following interfaces are also important in the definition of Ceylon's polymorphic operators:

- `Summable` supports the infix `+` operator,
- `Invertible` supports the prefix and infix `-` operators,
- `Ordinal` supports the unary `++` and `--` operators,
- `Numeric` supports the infix `*` and `/` operators,
- `Exponentiable` supports the power operator `^`,
- `Comparable` supports the comparison operators `<`, `>`, `<=`, `>=`, and `<=>`
- `Enumerable` supports the range operators `..` and `:`
- `Correspondence` supports the index operator,
- `Ranged` supports the subrange operators,
- `Boolean` is the basis of the logical operators `&&`, `||`, `!`, and
- `Set` is the basis of the set operators `|`, `&`, and, `~`

1.2.9. Comparison operators

In addition to the traditional `<`, `>`, `<=`, and `>=` operators, which evaluate to `Boolean`, there is a `<=>` operator, which produces an instance of the enumerated type `Comparison`.

```
switch(x<=>0)
case (smaller) {
    return sqrt(-x);
}
case (equal) {
    return 0;
}
case (larger) {
```

```
    return sqrt(x);
}
```

Two < or <= operators may be combined to determine if a value falls withing a range:

```
assert(0<quantity<=100);
```

1.2.10. Set operators

The operators | and & represent set union and intersection when they appear in a value expression. But, as we've already seen, when they appear in a type expression they represent type union and intersection! Indeed, there is a relationship between the two kinds of union/intersection:

```
Set<Integer> integers = ... ;
Set<Float> floats = ... ;
Set<Float|Integer> numbers = integers | floats;


Set<Foo> foos = ... ;
Set<Bar> bars = ... ;
Set<Foo&Bar> foobars = foos & bars;
```

The binary ~ operator represents complement (set subtraction).

1.2.11. Indexed operations

We can access an element of a Correspondence by using the index operator. Both `Lists` and `Maps` are instances of `Correspondence`:

```
"string must start with a \""
assert (exists ch = text[0], ch=='"');
```

All Lists are also instances of `Ranged`. We can produce a subrange of a `Ranged` by providing two endpoints:

```
if (text[i..i]=="/") {
    String,String] split = [text[...i-1], text[i+1...]];
    //...
}
```

We can also produce a subrange of a `Ranged` by providing a starting point and a length.

```
String selectedText = text[selection.offset:selection.length];
```

Please take careful note the difference between `..` and `:`, they have quite distinct purposes:

```
print("hello"[2..2]); //prints "l"
print("hello"[2:2]);  //prints "ll"

print("hello"[2..0]); //prints "leh"
print("hello"[2:0]);  //prints ""
```

1.3. Intro in functions and values

The two most basic constructs found in almost every programming language are functions and variables. In Ceylon, "variables" are, by default, assignable exactly ONCE. That is, they can't be assigned a new value after an initial value has been assigned. Therefore, we use the word value to talk about "variables" collectively, and reserve the word variable to mean a value which is explicitly defined to be reassignable.

```
String bye = "Adios";        //a value
variable Integer count = 0;  //a variable

bye = "Adieu";  //compile error
count = 1;      //allowed
```

Note that even a value which isn't a variable in this sense, may still be "variable" in the sense that its value varies between different runs of the program, or between contexts within a single execution of the program. A value may even be recalculated every time it is evaluated.

```
String name { return firstName + " " + lastName; }
```

If the values of firstName and lastName vary, then the value of name also varies between evaluations. A function takes this idea one step further. The value of a function depends not only upon the context in which it is evaluated, but also upon the arguments to its parameters.

```
Float sqr(Float x) { return x*x; }
```

In Ceylon, a value or function declaration can occur almost anywhere:

- as a toplevel, belonging directly to a package,
- as an attribute or method of a class,
- as a block-local declaration inside a different value or function body.

Indeed, as we'll see later, a value or function declaration may even occur inside an expression in some cases. Functions declarations look pretty similar to what you're probably already used to from other C-like languages, with two exceptions. Ceylon has:

- defaulted parameters
- variadic parameters

1.3.1. Defaulted parameters

A function parameter may specify a default value.

```
void hello(String name="World") {
    print("Hello, ``name``!");
}
```

Then we don't need to specify an argument to the parameter when we call the function:

```
hello(); //Hello, World!
hello("JBoss"); //Hello, JBoss!
```

Defaulted parameters must be declared after ALL required parameters in the parameter list of a function.

1.3.2. Variadic parameters

A variadic parameter of a function or class is declared using a postfix asterisk, for example, `String*`. There may be only one variadic parameter for a function or class, and it must be the last parameter.

```
void helloEveryone(String* names) {
    // ...
}
```

Inside the function body, the parameter names has type `[String*]`, a sequence type, which we'll learn about later. Thus, we can iterate the parameter using a for loop to get at the individual arguments.

```
void helloEveryone(String* names) {
     for (name in names) {
         hello(name);
     }
}
```

A nonempty variadic parameter is declared using a postfix plus sign, for example, `String+`. In this case, the caller must supply at least one argument.

```
void helloEveryone(String+ names) {
     for (name in names) {
          hello(name);
     }
}
```

To pass an argument to a variadic parameter we have three choices. We could: provide an explicit list of enumerated arguments, pass an iterable object producing the arguments, or specify a comprehension. The first case is easy:

```
helloEveryone("world", "mars", "saturn");
```

For the second case, Ceylon requires us to use the spread operator:

```
String[] everyone = ["world", "mars", "saturn"];
helloEveryone(*everyone);
```

We'll come back to the third case, comprehensions, later.

1.3.3. Fat arrows and forward declaration

Ceylon's expression syntax is much more powerful than Java's, and it's therefore possible to express a lot more in a single compact expression. So it's extremely common to encounter functions and values which simply evaluate and return an expression. So Ceylon lets us abbreviate such function and value definitions using a "fat arrow", `=>`. For example:

```
String name => firstName + " " + lastName;
```

Or:

```
Float sqr(Float x) => x*x;
```

Now's the time to get comfortable with this syntax, because you're going to be seeing quite a lot of it. Take careful note of the difference between a fat arrow:

```
String name => firstName + " " + lastName;
```

And an assignment:

```
String name = firstName + " " + lastName;
```

In the first example, the expression is recomputed every time name is evaluated. In the second example, the expression is computed once and the result assigned to name. We're even allowed to define a void function using a fat arrow. Earlier, we could have written `hello()` like this:

```
void hello() => print("Hello, World!");
```

In Java and C#, we're allowed to separate the declaration of a variable from the initialization of its value. We've already seen that this is also allowed in Ceylon. So we can write:

```
String name;
name = firstName + " " + lastName;
print(name);
```

But Ceylon even lets us do this with fat arrows:

```
String name;
name => firstName + " " + lastName;
print(name);
```

And even functions:

```
Float sqr(Float x);
sqr(Float x) => x*x;
print(sqr(0.01));
```

```
void hello();
hello() => print("Hello, World!");
hello();
```

The compiler makes sure we don't evaluate a value or invoke a function before assigning it a value or specifying its implementation, as we'll see later. (Because if we did, it would result in a `NullPointerException`, which Ceylon doesn't have!)

1.4. Control structures

Ceylon has six built-in control structures. There's nothing much new here for Java or C# developers, so a few quick examples without much additional commentary should suffice.

Gotcha! First, one "gotcha" for folks coming from other C-like languages: Ceylon doesn't allow you to omit the braces in a control structure. The following doesn't even parse:

```
if (x > 100) print("big");  //error
```

You are required to write:

```
if (x > 100) { print("big"); }
```

(The reason braces aren't optional in Ceylon is that an expression can begin with an opening brace, for example, `{"hello", "world"}`, so optional braces in control structures would make the whole grammar ambiguous.) OK, so here we go with the examples.

1.4.1. If conditionals

The if/else statement is totally traditional:

```
if (x > 1000) {
   print("really big");
}
else if (x > 100) {
   print("big");
}
else {
   print("small");
}
```

Later we will learn how if can narrow the type of references in its block. We've already seen an example of that, back when we talked about optional types. We often use the operators then and else instead of if.

1.4.2. Switch conditionals

The switch/case statement eliminates C's much-criticized "fall through" behavior and irregular syntax:

```
switch (x <=> 100)
case (smaller) { print("smaller"); }
case (equal) { print("one hundred"); }
case (larger) { print("larger"); }
```

The type of the switched expression may be an enumerated type, `String`, `Character`, or `Integer`. We'll have much more to say about switch when we discuss enumerated types.

1.4.3. Assertions

Ceylon also has an assert statement:

```
assert (length < 10);
```

Such assertions are good for making statements which you know have to be true, but are not apparent to other readers of the code (including the type checker!). Common uses of assert include things like preconditions, postconditions and class invariants. If the condition is false at runtime an exception is thrown. The exception message helpfully includes details of the condition which was violated, which is extra important when the assert has more than one condition.

```
assert (exists arg, !arg.empty);
```

To CUSTOMIZE the assertion message, add a doc annotation:

```
"length must be less than 10"
assert (length < 10);
```

Where applicable, the typechecker uses asserted type information when checking statements which follow the assertion, for example:

```
Integer? x = parseInteger("1");
assert (exists x);
// after the assert, x has type Integer instead of Integer?
value y = x + 10;
```

This is really just the same behavior we saw earlier, only this time it's happening in the middle of a block rather than at the start of an if block. (Don't worry, there's more on this later.) Note that, unlike Java's assert, which can be disabled at runtime, Ceylon's assertions are always enabled.

1.4.4. For loops

The for loop has an optional else block, which is executed when the loop completes normally, rather than via a return or break statement.

```
variable Boolean minors;
for (p in people) {
    if (p.age < 18) {
```

```
        minors = true;
        break;
    }
}
    else {
    minors = false;
}
```

There is no C-style for. Instead, you can use the lengthwise range operator `:` to produce a sequence of `Integers` given a starting point and a length:

```
for (i in min:len) { ... }
```

Alternatively, you can use the ordinary range operator `..` to produce a sequence of `Integers` given two endpoints:

```
for (i in min..max) { ... }
```

There are a couple of other tricks with for that we'll come back to later. We often use comprehensions or even higher order functions instead of for.

1.4.5. While loops

The while loop is traditional.

```
value it = names.iterator();
while (is String next = it.next()) {
    print(next);
}
```

There is no do/while statement.

1.4.6. Try statements

The try/catch/finally statement works just like Java's:

```
try {
message.send();
}
catch (ConnectionException|MessageException e) {
    tx.setRollbackOnly();
}
```

To handle all Ceylon exceptions, together with all JavaScript exceptions, or all Java exceptions that are subclasses of `java.lang.Exception`, we can catch the type `Exception` defined in `ceylon.language`. If we don't explicitly specify a type, `Exception` is inferred:

```
try {
    message.send();
}
catch (e) {  //equivalent to "catch (Exception e)"
    tx.setRollbackOnly();
}
```

To handle all exceptions, including subtypes of `java.lang.Error`, we can catch the root exception class `Throwable`. The `try` statement may optionally specify a "resource" expression, just like in Java. The resource is a `Destroyable` or an `Obtainable`.

```
try (Transaction()) {
    try (s = Session()) {
        s.persist(person);
    }
}
```

There are no Java-style checked exceptions in Ceylon.

1.4.7. Condition lists

Constructs like `if`, `while`, and `assert` accept a condition list. A condition list is simply an ordered list of multiple boolean, `exists`, `nonempty`, and `is` conditions. The condition list is satisfied if (and only if) every one of the conditions is satisfied. With plain `Boolean` conditions you could achieve the same thing with the `&&` operator of course. But a condition list lets you use the "structured typecasting" of `exists`, `is`, and friends in conditions appearing later in the same list. Let's see an example using `assert`:

```
value url = parserUri("http://ceylon-lang.org/download");
assert(exists authority = url.authority,
exists host = authority.hostname);
// do something with host
```

Here you can see two `exists` conditions in the `assert` statement, separated with a comma. The first one declares `authority` (which is inferred to be a String, rather than a `String?` because of the `exists`). The second condition then uses this in its own exists condition. The important thing to note is that the compiler lets us use `authority` in the second condition and knows that it's a `String`, not a `String?`. You can't do that by `&&`-ing multiple conditions together. You could do it by nesting several ifs, but that tends to lead to much less readable code, and doesn't work well in `while` statements or comprehensions.

1.5. Streams, sequences, and tuples

1.5.1. Iterables

An iterable object, or stream, is an object that produces a stream of values. Streams satisfy the interface Iterable. Ceylon provides some syntax sugar for working with streams: the type `Iterable<X,Null>` represents a STREAM that might not produce any values when it is iterated, and may be abbreviated `{X*}`, and the type `Iterable<X,Nothing>` represents a stream that always produces at least one value when it is iterated, and is usually abbreviated `{X+}`. We may construct an instance of `Iterable` using braces:

```
{String+} words = { "hello", "world" };
                      {String+} moreWords = { "hola", "mundo", *words };
```

The prefix `*` is called the spread operator. It "spreads" the values of a stream. So `moreWords` produces the values "hola", "mundo", "hello", "world" when iterated. As we'll see later, the braces may even contain a comprehension, making them much more powerful than what you see here. `Iterable` is a subtype of the interface `Category`, so we can use the in operator to test if a value is produced by the `Iterable`.

```
if (exists char = text[i],
    char in {',', '.', '!', '?', ';', ':'}) {
    //...
}
```

```
"index must be between 1 and 100"
assert (index in 1..100);
```

The `in` operator is just syntactic sugar for the method `contains()` of `Category`.

1.5.2. Iterating a stream

To iterate an instance of `Iterable`, we can use a for loop:

```
for (word in moreWords) {
    print(word);
}
```

(Note that in this context, the in keyword isn't the operator we just met above, it's just part of the syntax of the for loop.) If, for any reason, we need an index for each element produced by a stream, we can use a special variation of the for loop that is designed for iterating streams of `Entrys`:

```
for (i -> word in moreWords.indexed) {
    print("``i``: ``word``");
}
```

The indexed attribute returns a stream of entries containing the indexed elements of the original stream. (Note: the arrow `->` is syntax sugar for the class `Entry`. So we can write the type of the entry stream as `{<Integer->String>*}`.) It's often useful to be able to iterate two sequences at once. The `zipEntries()` function comes in handy here:

```
for (name -> place in zipEntries(names,places)) {
    print(name + " @ " + place);
}
```

1.5.3. Sequences

Some kind of array or list construct is a universal feature of all programming languages. The Ceylon language module

defines support for sequence types via the interfaces `Sequential`, `Sequence`, and `Empty`. Again, there is some syntax sugar associated with sequences:

- the type `Sequential<X>` represents a sequence that may be empty, and may be abbreviated `[X*]` or `X[]`,
- the type `Sequence<X>` represents a nonempty sequence, and may be abbreviated `[X+]`, and
- the type `Empty` represents an empty sequence and is abbreviated `[]`.

Some operations of the type `Sequence` aren't defined by `Sequential`, so you can't call them if all you have is `X[]`. Therefore, we need the `if (nonempty ... )` construct to gain access to these operations.

```
void printBounds(String[] strings) {
    if (nonempty strings) {
        //strings is of type [String+] here
        print(strings.first + ".." + strings.last);
    }
    else {
        print("Empty");
    }
}
```

Notice how this is just a continuation of the pattern established for null value handling. In fact, both these constructs are just syntactic abbreviations for type narrowing:

- `if (nonempty strings)` is an abbreviation for `if (is [String+] strings)`,
- just like `if (exists name)` is an abbreviation for `if (is Object name)`.

1.5.4. Sequence syntax sugar

There's lots more syntactic sugar for sequences. We can use a bunch of familiar Java-like syntax:

```
String[] operators = [ "+", "-", "*", "/" ];
String? plus = operators[0];
String[] multiplicative = operators[2..3];
```

Oh, and the expression `[]` evaluates to an instance of `Empty`.

```
[] none = [];
```

However, unlike Java, all these syntactic constructs are pure abbreviations. The code above is exactly equivalent to the following de-sugared code:

```
Sequential<String> operators = ... ;
Null|String plus = operators.get(0);
Sequential<String> multiplicative = operators.span(2,3);
```

(We'll come back to what the list of values in brackets means in a minute!) The `Sequential` interface extends `Iterable`, so we can iterate a `Sequential` using a `for` loop:

```
for (op in operators) {
    print(op);
}
```

1.5.5. Ranges

A `Range` is a kind of `Sequence`. The following:

```
Character[] uppercaseLetters = 'A'..'Z';
Integer[] countDown = 10..0;
```

Is just sugar for:

```
Sequential<Character> uppercaseLetters = Range('A','Z');
Sequential<Integer> countDown = Range(10,0);
```

In fact, this is just a sneak preview of the fact that almost all operators in Ceylon are just sugar for method calls upon a type. We'll come back to this later, when we talk about operator polymorphism. Ceylon doesn't need C-style for loops. Instead, combine for with the range operator:

```
variable Integer fac=1;
```

```
for (n in 1..100) {
    fac*=n;
    print("Factorial ``n``! = ``fac``");
}
```

1.5.6. Empty sequences and the bottom type

Finally, check out the definition of `Empty`. Notice that `Empty` is declared to be a subtype of `List<Nothing>`. This special type `Nothing`, often called the bottom type, represents:

- the empty set, or equivalently
- the intersection of all types.

Since the empty set is a subset of all other sets, Nothing is assignable to all other types. Why is this useful here? Well, `Correspondence<Integer,Element>` and `Iterable<Element>` are both covariant in the type parameter `Element`. So `Empty` is assignable to `Correspondence<Integer,T>` and `Iterable<T>` for any type `T`. That's why `Empty` doesn't need a type parameter.

Since there are no actual instances of `Nothing`, if you ever see an attribute or method of type `Nothing`, you know for certain that it can't possibly ever return a value. There is only one possible way that such an operation can terminate: by throwing an exception.

Another cool thing to notice here is the return type of the `first` and `item()` operations of `Empty`. You might have been expecting to see `Nothing?` here, since they override supertype members of type `T?`. But as we saw, `Nothing?` is just an abbreviation for `Null|Nothing`. And `Nothing` is the empty set, so the union `Nothing|T` of `Nothing` with any other type `T` is just `T` itself.

The Ceylon compiler is able to do all this reasoning automatically. So when it sees an `Iterable<Nothing>`, it knows that the operation first is of type `Null`, i.e. that its value is `null`.

1.5.7. Sequence gotchas for Java developers

Superficially, a sequence type looks a lot like a Java array, but really it's very, very different! First, of course, a sequence type `Sequential<String>` is an IMMUTABLE interface, it's not a mutable concrete type like an array. We can't set the value of an element:

```
String[] operators = .... ;
operators[0] = "^"; //compile error
```

Furthermore, the index operation `operators[i]` returns an optional type `String?`, which results in quite different code idioms. To begin with, we don't iterate sequences by index like in C or Java. The following code does not compile:

```
for (i in 0..operators.size-1) {
    String op = operators[i]; //compile error
    // ...
}
```

Here, `operators[i]` is a `String?`, which is not directly assignable to `String`. Instead, if we need access to the index, we use the special form of `for` shown above.

```
for (i -> op in entries(operators)) {
    // ...
}
```

Likewise, we don't usually do an upfront check of an index against the sequence length:

```
if (i>operators.size-1) {
    throw IndexOutOfBoundException();
}
else {
    return operators[i]; //compile error
}
```

Instead, we do the check after accessing the sequence element:

```
if (exists op = operators[i]) {
    return op;
}
else {
    throw IndexOutOfBoundException();
```

```
}
```

Indeed, this is a common use for `assert`:

```
 assert (exists op = operators[i]);
 return op;
```

We especially don't ever need to write the following:

```
if (i>operators.size-1) {
    return "";
}
else {
    return operators[i]; //compile error
}
```

This is much cleaner:

```
return operators[i] else "";
```

All this may take a little getting used to. But what's nice is that all the exact same idioms also apply to other kinds of `Correspondence`, including `Maps`.

1.5.8. Tuples

A tuple is a linked list which captures the static type of each individual element in the list. For example:

```
[Float,Float,String] point = [0.0, 0.0, "origin"];
```

This tuple contains a two `Floats` followed by a `String`. That information is captured in its static type, `[Float,Float,String]`. Each link of the list is an instance of the class `Tuple`. If you really must know, the code above is syntax sugar for the following:

```
Tuple<Float|String,Float,Tuple<Float|String,Float,Tuple<String,String>>>
point = Tuple(0.0, Tuple(0.0, Tuple("origin", [])));
```

Therefore, we always use syntax sugar when working with tuples. Tuple extends `Sequence`, so we can do all the usual kinds of sequency things to a tuple, iterate it, and so on. As with sequences, we can access a tuple element by index. But in the case of a tuple, Ceylon is able to determine the type of the element when the index is a literal integer:

```
Float x = point[0];
Float y = point[1];
String label = point[2];
Null zippo = point[3];
```

A unterminated tuple is a tuple where the last link in the list is a sequence, not an `Empty`. For example:

```
String[] labels = ... ;
[Float,Float,String*] point = [0.0, 0.0, *labels];
```

This tuple contains two `Floats` followed by an unknown number of `Strings`. Now we can see that a sequence type like `[String*]` or `[String+]` can be viewed as a degenerate tuple type!

1.6. Comprehensions

A comprehension is a convenient way to transform, filter, or combine a stream or streams of values before passing the result to a function. Comprehensions act upon, and produce, instances of `Iterable`. A comprehension may appear:

- inside braces, producing an iterable,
- inside brackets, producing a sequence,
- inside a positional argument list, as an argument to a variadic parameter, or
- inside a named argument list, as an iterable argument.

Comprehensions in iterable and sequence instantiation expressions The brace syntax for instantiating an iterable accepts a comprehension, so we can use a comprehension to transform any iterable:

```
{String*} names = { for (p in people) p.name };
```

Executing the above line of code doesn't actually do very much. In particular it doesn't actually iterate the collection

`people`, or evaluate the `name` attribute. That's because elements of the resulting `Iterable` are evaluated lazily. The bracket syntax for instantiating a sequence also accepts a comprehension, so we can use a comprehension to build a sequence:

```
String[] names = [ for (p in people) p.name ];
```

Since sequences are by nature immutable, executing the previous statement does iterate the people and evaluate their names. But it's best to think of that as the effect of the bracket syntax, not of the comprehension itself. Now, comprehensions aren't only useful for building iterables and sequences! They're a significantly more general purpose construct. The idea is that you can write a comprehension anywhere the language syntax accepts multiple values. That is to say, anywhere you could write a list of comma-separated expressions, or spread an iterable using `*`. (Aside: actually, we sometimes prefer think of the iterable instantiation syntax and sequence instantiation syntax as just a syntactic shorthand for an ordinary named argument instantiation expression. That's not precisely how the language specification defines these constructs, but it's a useful mental model to keep handy. So the idea is that anything we can write inside braces or brackets should also be syntactically legal inside a named argument list.)

Gotcha! It's important to have the right mental model of where a comprehension starts and finishes and what precisely it means. The comprehension is the bit which starts with for, and ends in an expression. The braces or brackets are not included. A comprehension produces multiple values, not a single value. Therefore a comprehension is not considered an expression and we can't directly assign a comprehension to a value reference! If we just need to store the iterable stream somewhere, without evaluating any of its elements, we can use an iterable instantiation expression, exactly like the one we've just seen.

Comprehensions as variadic arguments. One place where the language "accepts multiple values" is in the positional argument list for a function with a variadic parameter.

```
void printNames(String* names) => printAll(names, " and ");

printNames(for (p in people) p.name);
```

Arguments to variadic parameters are packaged into a sequence, so the comprehension is iterated eagerly, before the result is passed to the receiving function. Therefore, we don't usually use variadic parameters for processing streams in Ceylon. That's OK, because we have an alternative option that is designed precisely with stream processing in mind.

Comprehensions in named argument lists. Now let's see what makes comprehensions really useful. Suppose we had a class `HashMap`, with the following signature:

```
class HashMap<Key,Item>({Key->Item*} entries) { ... }
```

According to the previous chapter, we can pass multiple values to this parameter using a named argument list:

```
value numbersByName = HashMap { "one"->1, "two"->2, "three"->3 };
```

If multiple values are acceptable, so is a comprehension:

```
value numNames = ["one", "two", "three"];
value numbersByName = HashMap { for (i->w in numNames.indexed) w->i };
```

Going back to our previous example, we could construct a `HashMap<String,Person>` like this:

```
value peopleByName = HashMap { for (p in people) p.name->p };
```

As you've already guessed, the for clause of a comprehension works a bit like the for loop we met earlier. It takes each element of the Iterable stream in turn. But it does it lazily, when the receiving function actually iterates its argument! This means that if the receiving function never actually needs to iterate the entire stream, the comprehension will never be fully evaluated. This is extremely useful for functions like `every()` and `any()`:

```
if (every { for (p in people) p.age>=18 }) { ... }
```

The function `every()` in ceylon.language accepts a stream of `Boolean` values, and stops iterating the stream as soon as it encounters false. Now let's see what the various bits of a comprehension do.

1.6.1. Transformation

The first thing we can do with a comprehension is transform the elements of the stream using an expression to produce a new value for each element. This expression appears at the end of a comprehension. It's the thing that the resulting Iterable actually iterates! For example, this comprehension

```
for (p in people) p.name->p
```

results in an `Iterable<String->Person>`. For each element of `people`, a new `Entry<String,Person>` is constructed by

the `->` operator.

1.6.2. Filtering

The if clause of a comprehension allows us to skip certain elements of the stream. This comprehension produces a stream of numbers which are divisible by 3.

```
for (i in 0..100) if (i%3==0) i
```

It's especially useful to filter using `if (exists ...)`.

```
for (p in people) if (exists s = p.spouse) p->s
```

You can even use multiple if conditions:

```
for (p in people)
    if (exists s = p.spouse,
        nonempty inlaws = s.parents)
            p->inlaws
```

1.6.3. Products and joins

A comprehension may have more than one `for` clause. This allows us to combine two streams to obtain a stream of the values in their cartesian product:

```
for (i in 0..100) for (j in 0..10) Node(i,j)
```

Even more usefully, it lets us obtain a stream of associated values, a lot like a join in SQL.

```
for (o in orgs) for (e in o.employees) e.name
```

1.6.4. Comprehensions beginning in if

A comprehension may begin with an `if` clause. For example, this comprehension:

```
[ if (x>=0.0) x^0.5 ]
```

Produces the singleton `[2.0]` if `x==4.0`, and the empty tuple `[]` if `x<0.0`. Likewise, the comprehension:

```
{ if (exists list) for (x in list) x.string }
```

Produces an empty stream if `list` is `null`.

1.7. Functions

Folks who have a background in languages like ML might have expected that `void` would be identified with some "unit" type, for example, `Null`, or perhaps `[]`. But this approach would mean that a non-void method would not be able to refine a `void` method, and that a non-void function would not be able to be assigned to a void functional parameter. Therefore, perfectly reasonable code would be rejected by the compiler. Note that a void function with a concrete implementation implicitly returns the value `null`. This is different to a function declared to return the type `Anything`, which may `return` any value at all, but must do it explicitly, via a return statement. The following functions have the same type, `Anything()`, but don't do exactly the same thing:

```
Anything hello() {
    print("hello");
    return "hello";
}

void hello() {
    print("hello");
    //implicitly returns null
}
```

You shouldn't rely upon a function that is declared void returning `null`, because it might be a method that is refined by a non-void method, or a reference to a non-void function.

1.7.1. Defining higher order functions

We now have enough machinery to be able to write higher order functions. For example, we could create a `repeat()` function that repeatedly executes a function.

```
void repeat(Integer times,
    Anything(Integer) perform) {
        for (i in 1..times) {
            perform(i);
        }
}
```

Let's try it:

```
void printNum(Integer n) => print(n);
repeat(10, printNum);
```

Which would print the numbers 1 to 10 to the console. There's one problem with this. In Ceylon, as we'll see later, we often call functions using named arguments, but the `Callable` type does not encode the names of the function parameters. So Ceylon has an alternative, more elegant, syntax for declaring a parameter of type `Callable`:

```
void repeat(Integer times,
    void perform(Integer n)) {
        for (i in 1..times) {
            perform { n=i; };
        }
}
```

This version is also slightly more readable, so it's the preferred syntax.

1.7.2. Function references

When a name of a function appears without any arguments, like `printNum` does above, it's called a function reference. A function reference is the thing that really has the type `Callable`. In this case, printNum has the type `Callable<Anything,[Integer]>`, or simply `Anything(Integer)`. Now, remember how we said that `Anything` is both the return type of a `void` function, and also the logical root of the type hierarchy? Well that's useful here, since it means that we can assign a function with any return type to any parameter which expects a `void` function, as long as the parameter lists match:

```
Boolean attemptPrint(Integer n) {
    try {
        print(n);
        return true;
    }
    catch (Exception e) {
        return false;
    }
}
```

And call it like this

```
repeat(10, attemptPrint);
```

Another way we can produce a function reference is by partially applying a method to a receiver expression. For example, we could write the following:

```
class Hello(String name) {
    shared void say(Integer n) {
        print("Hello, ``name``, for the ``n``th time!");
    }
}
```

And call it like this:

```
repeat(10, Hello("Gavin").say);
```

Here the expression `Hello("Gavin").say` has the same type as print above. It is of type `Anything(Integer)`.

1.7.3. Static method and attribute references

A static method reference is a reference to a method, qualified by the type which declares the method.

```
value say = Hello.say;
```

In a static method reference, there's no receiving instance of the type! We have not provided any instance of `Hello` in this snippet. What we just wrote is quite different to this:

```
value say = Hello("Gavin").say;
```

So for a static method reference, to invoke the method, we must provide two items of information:

- first, an instance of the type, and
- then, the arguments of the function.

We provide these two items in distinct argument lists:

```
value say = Hello.say;
value sayHello = say(Hello("World"));
sayHello(3);
```

Or, simply:

```
value say = Hello.say;
say(Hello("World"))(3);
```

Let's fill in the types, to see what's really going on here:

```
Anything(Integer)(Hello) say = Hello.say;
Anything(Integer) sayHello = say(Hello("World"));
sayHello(3);
```

That is, a static function reference is a function that returns a function. A static attribute reference is a reference to an attribute, qualified by the type which declares the attribute.

```
Polar coord = .... ;
value angle = Polar.angle;
value radius = Polar.angle;
Float a = angle(coord);
Float radius = radius(coord);
```

Just to be sure, let's fill in the types:

```
Polar coord = .... ;
Float(Polar) angle = Polar.angle;
Float(Polar) radius = Polar.angle;
Float a = angle(coord);
Float radius = radius(coord);
```

Static attribute references work especially well with the `map()` method of `Iterable`:

```
{String*} names = people.map(Person.name);
```

1.7.4. Curried functions

A method or function may be declared in curried form, allowing the method or function to be partially applied to its arguments. A curried function has multiple lists of parameters:

```
Float adder(Integer n)(Float x) => x+n;
```

The `adder()` function has type `Float(Float)(Integer)`. We can invoke it with a single integer argument to get a reference to a function of type `Float(Float)`, and store this reference as a function, like this:

```
Float addOne(Float x);
addOne = adder(1);
```

Or as a value, like this:

```
Float(Float) addOne = adder(1);
```

(There only real difference between these two approaches is that in the first case we get to assign a name to the parameter of `addOne()`.) When we subsequently invoke `addOne()`, the actual body of `adder()` is finally executed, producing a `Float`:

```
Float three = addOne(2.0);
```

Gotcha! Did you notice that order of parameter lists is reversed in a type expression compared to a function declaration? Look again:

```
void printSum(String desc)(Float x, Float y) => print(desc + (x+y).string);
Anything(Float,Float)(String) printSumRef = printSum;
```

The order of parameter lists in the function declaration reflects the order in which we supply arguments when we invoke

the function. But in a function type expression, the return type comes always comes before the parameter types, so therefore the parameters which must be supplied first come at the last in the function type.

1.7.5. Anonymous functions

The most famous higher-order functions are a trio of functions for transforming, filtering, and summarizing sequences of values. In Ceylon, these three functions, `map()`, `filter()`, and `fold()` are methods of the interface `Iterable`. (They even have a fourth, slightly less glamorous friend called `find()`, also a method of `Iterable`.) As you've probably noticed, all the functions we've defined so far have been declared with a name, using a traditional C-like syntax. There's nothing wrong with passing a named function to `map()` or `filter()`, and indeed that is often useful:

```
Float max = measurements.fold(0.0)(largest<Float>);
```

However, quite commonly, it's inconvenient to have to declare a whole named function just to pass it to `map()`, `filter()`, `fold()` or `find()`. Instead, we can declare an anonymous function inline, as part of the argument list:

```
Float max = measurements.fold(0.0)
                    ((Float max, Float num) =>
                        num>max then num else max);
```

An anonymous function has:

- optionally, the keyword `function` or `void`, and then
- a parameter list, enclosed in parentheses, followed by
- a fat arrow, `=>`, with an expression, or
- a block.

So we could rewrite the above using a block

```
Float max = measurements.fold(0.0)
                    ((Float max, Float num) {
                        return num>max then num else max;
                    });
```

Note that it's quite difficult to come up with a good way to format anonymous functions with blocks, so it's usually better to just give the function a name and use it by reference.

1.7.6. Anonymous function parameter type inference

When an anonymous function occurs in an argument list, it's usually, but not always, possible to infer the parameter types. We can write the previous example as follows:

```
Float max = measurements.fold(0.0)
    ((max, num) => num>max then num else max);
```

Gotcha! You might have noticed that `fold()` is defined in curried form, with two argument lists. The reason it's defined like this is to allow inference of the parameter types of its function argument. If it weren't defined in curried form, we would have to explicitly specify the anonymous function parameter types, since the type of the first argument of `fold()` would not be taken into account when inferring parameter types.

1.7.7. Composition and curry

The function `compose()` performs function composition. For example, given the functions `print()` and `plus()` in `ceylon.language`, with the following signatures:

```
shared void print(Anything line) { ... }
```

```
shared Value plus<Value>(Value x, Value y)
    given Value satisfies Summable<Value> { ... }
```

We can see that the type of the function reference print is `Anything(Anything)`, and that the type of the function reference `plus<Float>` is `Float(Float,Float)`. Then we can write the following:

```
Anything(Float,Float) printSum = compose(print,plus<Float>);
    printSum(2.0,2.0); //prints 4.0
```

The function `curry()` produces a function with multiple parameter lists, given a function with multiple parameters:

```
Anything(Float)(Float) printSumCurried = curry(printSum);
Anything(Float) printPlus2 = printSumCurried(2.0);
printPlus2(2.0); //prints 4.0
```

The function `uncurry()` does the opposite, giving us back our original uncurried signature:

```
Anything(Float,Float) printSumUncurried = uncurry(printSumCurried);
```

Note that `compose()`, `curry()`, and `uncurry()` are ordinary functions, written in Ceylon.

1.7.8. The spread operator

We've already seen a few examples of the spread operator. We've seen how to use it to instantiate an iterable:

```
{ "hello", *names }
```

Or a tuple:

```
[x, y, *labels]
```

We can also use it when calling a function. Consider the following function:

```
String formatDate(String format,
                Integer day,
                Integer|String month,
                Integer year) {
                ...
}
```

And suppose we have a tuple representing a date:

```
value date = [15, "January", 2010];
```

Then we can pass the date to our function like this:

```
formatDate("dd MMMMM yyyy", *date)
```

Notice that the type of the tuple `["dd MMMMM yyyy", *date]` is:

```
[String,Integer,String,Integer]
```

Now consider type of the function `formatDate`. It is:

```
String(String,Integer,Integer|String,Integer)
```

Or rather:

```
Callable<String,[String,Integer,Integer|String,Integer]>
```

Since the tuple type `[String,Integer,String,Integer]` is a subtype of `[String,Integer,Integer|String,Integer]`, the invocation is well-typed. This demonstrates the relationship between tuples and function argument!

1.8. Types

1.8.1. Narrowing the type of an object reference

In any language with subtyping there is the (hopefully) occasional need to perform narrowing conversions. In most statically-typed languages, this is a two-part process. For example, in Java, we first test the type of the object using the instanceof operator, and then attempt to downcast it using a C-style typecast. This is quite curious, since there are virtually no good uses for instanceof that don't involve an immediate cast to the tested type, and typecasts without type tests are dangerously non-typesafe.

As you can imagine, Ceylon, with its emphasis upon static typing, does things differently. Ceylon doesn't have C-style typecasts. Instead, we must test and narrow the type of an object reference in one step, using the special `if (is ... )` construct. This construct is very, very similar to `if (exists ... )` and `if (nonempty ... )`, which we met earlier.

```
void printIfPrintable(Object obj) {
    if (is Printable obj) {
        obj.printObject();
    }
}
```

There's also a special `if (!is ... )` construct which comes in handy from time to time. The switch statement can be used in a similar way:

```
void switchingPrint(Object obj) {
    switch(obj)
    case (is Hello) {
        obj.printMsg();
    }
    case (is Person) {
        print(obj.firstName);
    }
    else {
        print(obj.string);
    }
}
```

These constructs protect us from inadvertantly writing code that would cause a `ClassCastException` in Java, just like `if (exists ... )` protects us from writing code that would cause a `NullPointerException`. The `if (is ... )` construct actually narrows to an intersection type.

1.8.2. Intersection types

An expression is assignable to an intersection type, written `X&Y`, if it is assignable to both `X` and `Y`. For example, since `Tuple` is a subtype of `Iterable` and of `Correspondence`, the tuple type `[String,String]` is also a subtype of the intersection `{String*} & Correspondence<Integer,String>`. The supertypes of an intersection type include all supertypes of every intersected type. Therefore, the following code is well-typed:

```
{String*} & Correspondence<Integer,String> strings
    = ["hello", "world"];
String? str = strings.get(0);  //call get() of Correspondence
Integer size = strings.size;  //call size of Iterable
```

Now consider this code, to see the effect of `if (is ...)`:

```
{String*} strings = ["hello", "world"];
if (is Correspondence<Integer,String> strings) {
    //here strings has type
    //{String*} & Correspondence<Integer,String>
    String? str = strings.get(0);
    Integer size = strings.size;
}
```

Inside the body of the if construct, `strings` has the type `{String*} & Correspondence<Integer,String>`, so we can call operations of both `Iterable` and of `Correspondence`.

1.8.3. Union types

An expression is assignable to a union type, written `X|Y`, if it is assignable to either `X` or `Y`. The type `X|Y` is always a supertype of both `X` and `Y`. The following code is well-typed:

```
void printType(String|Integer|Float val) { ... }

printType("hello");
printType(69);
printType(-1.0);
```

But what operations does a type like `String|Integer|Float` have? What are its supertypes? Well, the answer is pretty intuitive: T is a supertype of `X|Y` if and only if it is a supertype of both `X` and `Y`. The Ceylon compiler determines this automatically. So the following code is also well-typed:

```
Integer|Float x = -1;
Number num = x;  // Number is a supertype of both Integer and Float
String|Integer|Float val = x; // String|Integer|Float is a supertype of Integer|Float
Object obj = val; // Object is a supertype of String, Integer, and Float
```

However, the following code is not well-typed, since `Number` is not a supertype of `String`.

```
String|Integer|Float x = -1;
Number num = x; //compile error: String is not a subtype of Number
```

Of course, it's very common to narrow an expression of union type using a `switch` statement. Usually, the Ceylon compiler forces us to write an `else` clause in a `switch`, to remind us that there might be additional cases which we have not handled. But if we exhaust all cases of a union type, the compiler will let us leave off the else clause.

```
void printType(String|Integer|Float val) {
```

```
switch (val)
case (is String) { print("String: ``val``"); }
case (is Integer) { print("Integer: ``val``"); }
case (is Float) { print("Float: ``val``"); }
}
```

A union type is a kind of enumerated type.

Gotcha! The cases of a switch statement must be disjoint. Since `String`, `Integer`, and `Float` are disjoint types, the above `switch` statement is legal. If a union type is formed from types which aren't disjoint, those types can't be used as distinct `cases`.

1.8.4. More about disjointness

As we've seen, disjointness is a useful property for two types to have, since it lets us use them as cases of the same switch statement. Therefore, the compiler expends some effort to determine if two types are disjoint. For example: if `X` and `Y` are classes, `X` is not a subclass of `Y`, and `Y` is not a subclass of `X`, then `X` and `Y` are disjoint, if `X` is a final class and `Y` is an interface not satisfied by `X`, then `X` and `Y` are disjoint, and two instantiations of a generic type may be disjoint, for example, `MutableList<String>` and `MutableList<Integer>`. (There's much more information about disjointness in the spec.) When the compiler encounters an intersection type involving disjoint types, for example, `String&Integer`, it automatically simplifies this type to the bottom type `Nothing`.

1.9. Classes

1.9.1. Identifier naming

The case of the first character of an identifier is significant. Type (interface, class, and type parameter) names must start with an initial capital letter. Function and value names start with an initial lowercase letter or underscore. The Ceylon compiler is very fussy about this. You'll get a compilation error if you write:

```
class hello() { ... } //compile error
```

or:

```
String Name = .... //compile error
```

There is a way to work around this restriction, which is mainly useful when calling legacy Java code. You can "force" the compiler to understand that an identifier is a type name by prefixing it with `\I`, or that it is a function or value name by prefixing it with `\i`. For example, `\iRED` is considered an initial lowercase identifier. So the following declarations are acceptable, but definitely not recommended, except in the interop scenario:

```
class \Ihello() { ... } //OK, but not recommended
```

and:

```
String \iName = .... //OK, but not recommended
```

1.9.2. Creating your own class

Our first class is going to represent points in a polar coordinate system. Our class has two parameters, two methods, and an attribute.

```
"A polar coordinate"
class Polar(Float angle, Float radius) {

    shared Polar rotate(Float rotation)
    => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
    => Polar(angle, radius*dilation);

    shared String description
    = "(``radius``,``angle``)";
}
```

There's two things in particular to notice here:

- The parameters used to instantiate a class are specified as part of the class declaration, right after the name of the class. There's no Java-style constructors in Ceylon. This syntax is less verbose and more regular than Java, C#, or C++.

- We make use of the parameters of a class anywhere within the body of the class. In Ceylon, we often don't need to define explicit members of the class to hold the parameter values. Instead, we can access the parameters `angle` and `radius` directly from the `rotate()` and `dilate()` methods, and from the expression which specifies the value of `description`.

Notice also that Ceylon doesn't have a new keyword to indicate instantiation, we just "invoke the class", writing `Polar(angle, radius)`. The `shared` annotation determines the accessibility of the annotated type, attribute, or method. Before we go any further, let's see how we can hide the internal implementation of a class from other code.

1.9.3. Hiding implementation details

Ceylon doesn't make a distinction between `public`, `protected` and "default" visibility like Java does. Instead, the language distinguishes between:

- program elements which are visible only inside the scope in which they are defined, and

- program elements which are visible wherever the thing they belong to (a type, package, or module) is visible.

By default, members of a class are hidden from code outside the body of the class. By annotating a member with the `shared` annotation, we declare that the member is visible to any code to which the class itself is visible. And, of course, a class itself may be hidden from other code. By default, a toplevel class is hidden from code outside the package in which the class is defined. Annotating a top level class with `shared` makes it visible to any code to which the package containing the class is visible. Finally, packages are hidden from code outside the module to which the package belongs by default. Only explicitly shared packages are visible to other modules.

1.9.4. Exposing parameters as attributes

If we want to expose the `angle` and `radius` of our `Polar` coordinate to other code, we need to define attributes of the class. It's very common to assign parameters of a class directly to a `shared` attribute of the class, so Ceylon provides a streamlined syntax for this.

```
"A polar coordinate"
class Polar(angle, radius) {

    shared Float angle;
    shared Float radius;

    shared Polar rotate(Float rotation)
    => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
    => Polar(angle, radius*dilation);

    shared String description
    = "(``radius``,``angle``)";

}
```

Code that uses `Polar` can access the attributes of the class using a very convenient syntax.

```
Cartesian cartesian(Polar polar)
=> Cartesian(polar.radius*cos(polar.angle),
polar.radius*sin(polar.angle));
```

There's an even more compact way to write the code above, though it's often less readable:

```
"A polar coordinate"
class Polar(shared Float angle, shared Float radius) {

    shared Polar rotate(Float rotation)
    => Polar(angle+rotation, radius);

    shared Polar dilate(Float dilation)
    => Polar(angle, radius*dilation);

    shared String description
    = "(``radius``,``angle``)";
}
```

This illustrates an important feature of Ceylon: there is almost no essential difference, aside from syntax, between a para-

meter of a class, and a value declared in the body of the class. Instead of declaring the attributes in the body of the class, we simply annotated the parameters `shared`. We encourage you to avoid this shortcut when you have more than one or two parameters.

1.9.5. Initializing attributes

The attributes `angle` and `radius` are references, the closest thing Ceylon has to a Java field. Usually we specify the value of a reference when we declare it.

```
Float x = radius * sin(angle);
String greeting = "Hello, ``name``!";
Integer months = years * 12;
```

On the other hand, it's sometimes useful to separate declaration from assignment.

```
shared String description;
if (exists label) {
    description = label;
}
else {
    description = "(``radius``,``angle``)";
}
```

But if there's no constructors in Ceylon, where precisely should we put this code? We put it directly in the body of the class!

```
"A polar coordinate with an optional label"
class Polar(angle, radius, String? label) {

    shared Float angle;
    shared Float radius;

    shared String description;
    if (exists label) {
        description = label;
    }
    else {
        description = "(``radius``,``angle``)";
    }

    // ...

}
```

The Ceylon compiler forces you to specify a value of any reference before making use of the reference in an expression.

```
Integer count;
void inc() {
    count++;   //compile error
}
```

1.9.6. Abstracting state using attributes

If you're used to writing JavaBeans, you can think of a reference as a combination of several things:

- a field,
- a getter, and, sometimes,
- a setter.

That's because not every value is a reference like the one we've just seen; others are more like a getter method, or, sometimes, like a getter and setter method pair. We'll need to expose the equivalent cartesian coordinates of a `Polar`. Since the cartesian coordinates can be computed from the polar coordinates, we don't need to define state-holding references. Instead, we can define the attributes as getters.

```
import ceylon.math.float { sin, cos }

"A polar coordinate"
class Polar(angle, radius) {

    shared Float angle;
    shared Float radius;

    shared Float x => radius * cos(angle);
    shared Float y => radius * sin(angle);
```

```
    // ...

}
```

Notice that the syntax of a getter declaration looks a lot like a method declaration with no parameter list. So in what way are attributes "abstracting state"? Well, code that uses `Polar` never needs to know if an attribute is a reference or a getter. Now that we know about getters, we could rewrite our `description` attribute as a getter, without affecting any code that uses it.

```
"A polar coordinate, with an optional label"
class Polar(angle, radius, String? label) {

    shared Float angle;
    shared Float radius;

    shared String description {
        if (exists label) {
            return label;
        }
        else {
            return "(``radius``,``angle``)";
        }
    }
}
```

1.9.7. Living without static members

Right at the beginning of the tour, we mentioned that Ceylon doesn't have `static` members like in Java, C#, or C++. Instead of a `static` member, we either:

- use a toplevel function or value declaration, or
- in the case where several "static" declarations need to share some private stuff, members of a singleton object declaration.

The lack of static members results in a minor gotcha for newcomers.

Gotcha! The syntax `Polar.radius` is legal in Ceylon, and we even call it a static reference, but it does not usually mean what you think it means! Sure, if you're taking advantage of Ceylon's Java interop, you can call a `static` member of a Java class using this syntax, just like you would in Java:

```
import java.lang { Runtime }

Integer procs = Runtime.runtime.availableProcessors();
```

Or, alternatively, you could write the following, directly importing the static member:

```
import java.lang { Runtime { runtime } }

Integer procs = runtime.availableProcessors();
```

But in regular Ceylon code, an expression like `Polar.radius` is not a reference to a static member of the class `Polar`. We'll come back to the question of what a "static reference" really is, when we discuss higher-order functions.

1.9.8. Living without overloading

It's time for some bad news: Ceylon doesn't have method or constructor overloading (the truth is that overloading is the source of various problems in Java, especially when generics come into play). However we can emulate most non-harmful uses of constructor or method overloading using: defaulted parameters, variadic parameters (varargs), and union types or enumerated type constraints. We're not going to get into all the details of these workarounds right now, but here's a quick example of each of the three techniques:

```
//defaulted parameter
void println(String line, String eol = "\n")
    => process.write(line + eol);


//variadic parameter
void printlns(String* lines) {
    for (line in lines) {
        println(line);
    }
```

```
}


//union type
void printName(String|Named name) {
    switch (name)
    case (is String) {
        println(name);
    }
    case (is Named) {
        println(name.first + " " + name.last);
    }
}
```

Don't worry if you don't completely understand the third example just yet, we'll come back to it later in the tour. Let's make use of this idea to "overload" the "constructor" of `Polar`.

```
"A polar coordinate with an optional label"
class Polar(angle, radius, String? label=null) {

shared Float angle;
shared Float radius;

shared String description {
    if (exists label) {
        return label;
    }
    else {
        return "(``radius``,``angle``)";
    }
}

// ...

}
```

Now we can create `Polar` coordinates with or without labels:

```
Polar origin = Polar(0.0, 0.0, "origin");
Polar coord = Polar(r, theta);
```

Later, we'll learn about named arguments, which we often use to make instantiation expressions more readable, especially when the class has more than two parameters:

```
Polar origin = Polar { angle = 0.0; radius = 0.0; label = "origin"; };
Polar coord = Polar { radius = r; angle = theta; };
```

Gotcha! Even with these "emulation" techniques, not every case of a legal overloaded Java method can be represented directly in Ceylon. In such situations it's necessary to exert a little more effort to come up with distinct names.