Introducing the Ceylon Project

Gavin King
Red Hat
`in.relation.to/Bloggers/Gavin`

About this session

- I’m going to talk about why we started work on this project
- I’m going to cover some basic examples at a very shallow level
- I’m not going to get into the details of the type system
- If you’re interested, come to my second presentation: “The Ceylon Language”
- This project is not yet available to the public and has not even been officially announced
    - QCon China is getting a special sneak preview - the first time I’m talking about the project in public!

Why we’re (still) fans of Java

- Java was the first language to feature the following “perfect” combination of features:
    - virtual machine execution, giving platform independence
    - automatic memory management and safe referencing
    - static typing
    - lexical scoping
    - readable syntax
- Therefore, Java was the first language truly suitable for
    - large team development, and
    - large-scale deployments of multi-user applications.
- It turns out that large teams developing multi-user applications describes the most interesting class of project in business computing

Why we’re (still) fans of Java

- Java is easy
    - Java’s syntax is rooted in standard, everyday mathematical notion taught in high schools and used by mathematicians, engineers, and software developers
        - not the lambda calculus used only by theoretical computer scientists
    - The language is mostly simple to learn and the resulting code is extremely easy to read and understand
    - Static typing enables sophisticated tooling including automatic refactoring, code navigation, and code completion
        - this kind of tooling is simply not possible without static typing
- Java is robust
    - With static typing, automatic memory management, and no C-style pointers, most bugs are found at development time

Why we’re (still) fans of Java

- The Java community is made of ordinary people trying to solve practical problems
    - Java is unashamedly focussed on problems relevant to business computing
    - The culture is a culture of openness that rejects dominance by any single company or interest
    - Java has remained committed to platform independence and portability
    - The community has a huge tradition of developing and sharing reusable code (frameworks, libraries)

Why we're frustrated

- After ten often-frustrating years developing frameworks for Java, we simply can't go any further without a better solution for defining structured data and user interfaces
    - Java is joined at the hip with XML, and this hurts almost every Java developer almost every day
    - There is simply no good way to define a user interface in Java, and that is a *language* problem
- Lack of a language-level modularity solution resulted in the creation of monstrous, over-complex, harmful technologies like Maven and OSGi.
    - Instead of modules, Java has multiple platforms, which has divided the developer community
- Lack of support for first-class and higher-order functions results in much unnecessary verbosity in everyday code
- Meta-programming in Java is clumsy and frustrating, reducing the quality of framework and other generic code

Why we’re frustrated

- A number of other “warts” and mistakes annoy us every day, for example
  - getters/setters
  - arrays and primitive types
  - non-typesafety of null values
  - the dangerous `synchronized` keyword
  - clumsy annotation syntax
  - verbose constructor syntax
  - broken == operator
  - checked exceptions
  - complex parametric polymorphism system (generics) that few developers completely understand
  - ad-hoc (broken?) block structure
  - clumsy, error-prone `instanceof` and typecast syntax

Why we’re frustrated

- Most of all, we’re frustrated by the SE SDK
  - designed in haste 15 years ago, and never properly modernized, it still has an experimental, work-in-progress feel about it
  - but is simultaneously bloated with obscure stuff
  - features some truly bizarre things
    - e.g. all Java objects are semaphores ?!
  - many basic tasks are absurdly difficult to accomplish
    - e.g. anything involving `java.io` or `java.lang.reflect`
  - overuses stateful (mutable) objects
    - especially the highly overrated collections framework

The Ceylon Project

- What would a language and SDK for business computing look like if it were designed today, with an eye to the successes and failures of the Java language and Java SE SDK?

The Ceylon Project

- This much is clear:
  - It would run on the Java Virtual Machine
  - It would feature static typing
  - It would feature automatic memory management and safe referencing
  - It would retain Java's readability
  - It would feature first-class and higher-order functions
  - It would provide a declarative syntax for defining user interfaces and structured data
  - It would feature built-in modularity
  - It would strive to be easy to learn and understand

The Ceylon Project

- Unfortunately, there’s no existing language that truly fits these requirements
- My team has spent the past two years designing what we think the language should look like, writing a language specification, an ANTLR grammar, and a prototype compiler
  - You can’t write code in the language just yet!
  - We plan an initial release of the compiler later this year
- I can’t cover the whole language, or even explain the most interesting principles and concepts in the short time I have here
  - The most I can do is give a taste of what some code looks like

Hello World

put this in a file called `hello.ceylon`

```
void hello() {
    writeLine(“Hello World!”);
}
```

The language has a strict recursive, regular block structure governing visibility and lifecycle of declarations. Therefore, there’s no equivalent of Java’s `static`. Instead, a toplevel method declaration fills a similar role.

Hello World

API documentation is specified using annotations.

```
doc “The classic Hello World program”
by “Gavin”
void hello() {
    writeLine(“Hello World!”);
}
```

Modifiers like `abstract`, `variable`, `shared`, `deprecated` aren’t keywords, they’re just annotations.

Hello World

`void` *is* a keyword!

```
void hello(String name) {
    writeLine(“Hello “ name ”!”);
}
```

String interpolation has a simple syntax - very useful in user interface definitions.

Hello World

Defaulted parameters are optional.

```
void hello(String name = ”World”) {
    writeLine(“Hello “ name ”!”);
}
```

Defaulted parameters are extremely useful, since Ceylon does not support method overloading (or any other kind of overloading).

Hello World

If a value of type `T` can be null, it must be declared as type `Optional<T>`, which may be abbreviated to `T?`.

```
void hello() {
    String? name = process.args.first;
    if (exists name) {
        writeLine(“Hello “ name ”!”);
    }
    else {
        writeLine(“Hello World!”);
    }
}
```

Use of an optional value must be guarded by the `if (exists ... )` construct. Therefore, `NullPointerExceptions` are impossible.

Classes

All values are instances of a class.

```
class Counter() {
    variable Natural count := 0;
```

Attributes and local variables are immutable by default. Assignable values must be annotated `variable`.

```
    shared void increment() {
        count++;
    }

}
```

The `shared` annotation makes a declaration visible outside the block in which it is defined. By default, any declaration is block local.

Classes

```
class Counter() {
    variable Natural count := 0;
    shared void increment() {
        count++;
    }
    shared Natural currentValue {
        return count;
    }
}
```

A *getter* looks like a method without a parameter list.

An attribute *may be a simple value, a getter, or a getter/setter pair.*

Classes

There is no `new` keyword.

```
Counter c = Counter();
c.increment();
writeLine(c.currentValue);
```

Attribute getters are called just like simple attributes. The client doesn't care what type of attribute it is.

Attributes are polymorphic. A subclass may override a superclass attribute. It may even override a simple attribute with a getter or vice versa!

Classes

The `local` keyword may be used in place of a type for block-local declarations.

```
local c = Counter();
c.increment();
writeLine(c.currentValue);
```

You can't use `local` for `shared` declarations. One consequence of this is that the compiler can do type inference in a single pass of the code!

Classes

```
class Counter() {
    variable Natural count := 0;
    ...
    shared Natural currentValue {
        return count;
    }
    shared assign currentValue {
        count := currentValue;
    }
}
```

Assignment to a `variable` value or attribute setter is done using the `:=` operator. The `=` specifier is used only for specifying immutable values.

Classes

There is no constructor syntax. Instead, the class itself declares parameters, and the body of the class may contain initialization logic.

```
class Counter(Natural initialValue) {
    if (initialValue>1000) {
        throw OutOfRangeException();
    }
    variable Integer count := initialValue;
    ...
}
```

How can a class have multiple constructors? It can’t! There’s no overloading in Ceylon.

Sequences

Sequences are immutable objects that are a bit like arrays.

```
Sequence<String> itin =
        Sequence(“Guanajuato”, “Mexico”,
           “Vancouver”, “Auckland”,
           “Melbourne”);

String? mex = itin.value(1);
Sequence<String> layovers =
        itin.range(1..3);

Sequence<String> longer = join(itin,
        Sequence(“Hong Kong”, “Beijing”));
```

Sequences

Syntactic abbreviations allow us to eliminate the verbosity.

```
String[] itin =
        { “Guanajuato”, “Mexico”,
           “Vancouver”, “Auckland”,
           “Melbourne” };

String? mex = itin[1];
String[] layovers =
        itin[1..3];

String[] longer = itin +
        { “Hong Kong”, “Beijing” };
```

Higher-order functions

A parameter may be a method signature, meaning that it accepts references to methods.

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

The “functional” parameter may be invoked just like any other method.

Higher-order functions

```
repeat(3, hello);
```

A reference to a method is just the name of the method, without an argument list.

Higher-order functions

```
repeat(3, person.sayHello);
```

We can even “curry” the method receiver.

Higher-order functions

We may define a method “by reference”.

```
void hello(String name) = hello;
```

The name of the method, without arguments, refers to the method itself.

```
void hello2(String name) = person.sayHello;
```

Unlike other languages with first-class functions, Ceylon doesn’t have a syntax for anonymous functions (“lambdas”) that appear in expressions.

Higher-order functions

```
repeat(3)
perform() {
    writeLine(“Hola Mundo!”);
};
```

The method name

A parameter name

Alternatively, a method may be defined inline, as part of the invocation. This syntax is stolen from Smalltalk.

Higher-order functions

```
repeat(3)
perform {
    writeLine(“Hola Mundo!”);
};
```

We may omit the empty parameter list.

This allows a library to define syntax for new control structures, assertions, comprehensions, etc.

Higher-order functions

A method may declare multiple lists of parameters. The method body is executed after arguments have been supplied to all parameter lists.

```
Float add(Float x)(Float y) {
    return x+y;
}
```

Higher-order functions

We can “curry” a list of arguments.

```
Float addOne(Float y) = add(1.0);
Float three = addOne(2.0);
```

Providing arguments to just one parameter list produces a method reference.

The point of all this is that we are able to provide all the functionality of first-class and higher-order functions without needing to resort to unnatural syntactic constructs inspired by the lambda calculus notation.

Closure

An inner declaration always has access to parameters, locals, and attributes of the containing declaration.

```
void aMethod(String name) {
    void hello() {
        writeLine(“Hello “ name “!”);
    }
}
```

Notice how regular *the language syntax is!*

```
class AClass(String name) {
    void hello() {
        writeLine(“Hello “ name “!”);
    }
}
```

Named argument syntax

```
String join(String separator,
            String... strings) { ... }


join(“, ”, “C”, “Java”, Smalltalk”);

join { separator = ”, “;
    “C”, “Java”, Smalltalk” };
```

A named argument invocation is enclosed in braces, and non-vararg arguments are listed using the `name=value;` syntax.

Higher-order functions and named arguments

```
repeat {
    times = 3;
    void perform() {
        writeLine(“Hola Mundo!”);
    }
};
```

The method name

A parameter name

Another parameter name

A named argument may even be a method definition.

Named argument syntax

```
Html hello {
    Head head { title = “Squares”; }
    Body body {
        Div {
            cssClass = “greeting”;
            “Hello” name “!”
        }
    }
}
```

This looks like a typesafe declarative language (for example XML) with built-in templating. But it’s actually written in a general-purpose language!

Named argument syntax

```
class Table(String title, Natural rows,
    Column... columns) { ... }

class Column(String heading,
    String content(Natural row)) { ... }
```

We can define the “schema” of a declarative language as a set of classes.

Named argument syntax

```
Table squares {
    title = “Squares”;
    rows = 10;
    Column {
        heading = “x”;
        String content(Natural row) {
            return $row;
        }
    }
    Column {
        heading = “x**2”;
        String content(Natural row) {
            return $row**2;
        }
    }
}
```

Notice the use of callback methods!

What next?

- If you’re interested to learn more, come to the next talk “The Ceylon Language”
- We need help implementing the compiler and designing the SDK.
- This isn’t worth doing unless we do it as a community!

Questions?