Ceylon at vJUG

Gavin King - Red Hat
profiles.google.com/gavin.king
ceylon-lang.org
We’re fans of Java
If I sound critical of Java (or any other language) in this presentation it’s merely to identify problems that require solutions.

Indeed, a lot of criticism of Java is IMO deeply misplaced—but that doesn’t mean there’s nothing wrong with Java!
What is it?

A programming language:

- That runs on *virtual machines*
- To be specific, the Java VM, and JavaScript VMs
- Defined by a *specification*
- With a syntax that looks conventional but is actually very flexible
- With an extremely powerful and elegant type system
- With built-in *modularity*
- With its own language module and SDK
- And *excellent tooling*
What is it?

Where it runs:

• On Java SE, with the Ceylon module runtime
• In any OSGi container: Eclipse, Apache Felix, WildFly, GlassFish, ...
• On Vert.x
• On Node.js
• In a web browser, with Common JS Modules (require.js)
• In a Java servlet engine, via ceylon war (in the next release)
What about interop?

Interoperable with native code

- It can be used to build a cross-platform module that executes in both virtual machine environments, depending only on other cross-platform modules written in pure Ceylon.

- Or, it can be used to write a module that targets only one of the two virtual machines and interoperates with native Java or JavaScript code for that platform.

- Interoperation with JavaScript is via dynamic typing, or via writing an interface in Ceylon that ascribes static types to the JS API.
A few unique things

A platform-neutral language module

- *Designed for multiplatform use*—the language and language module completely abstract the details of the virtual machine
- *Reified generics*, along with a *typesafe metamodel* that provides access to generic type arguments at runtime
- *Union and intersection types*—the foundation for unambiguous *type inference* and *flow-sensitive typing*
- Representation and abstraction of *function and tuple types* within the type system—without an explosion of single-method interface types or Function1, Function2, Function3, ...
- A simple, unified type system, with elegant *syntax sugar* that helps reduce verbosity without harming readability
Idiom #1

Idiom: functions with multiple outcomes

For example, an operation might return a File, a Url, or nothing:

```java
//Java
Object parsePath(String path)
    throws SyntaxException { ... }
```

We can handle the different outcomes using `instanceof`, type casts, and catch:

```java
try {
    Object result = parsePath(path);
    if (result instanceof File) {
        File file = (File) result;
        return lines(file);
    }
    if (result instanceof Url) {
        Url url = (Url) result;
        return new Request(url).execute().getContent().getLines();
    }
}
catch (SyntaxException se) { return emptyList(); }
```
Idiom #1

**Idiom: functions with multiple outcomes**

A function with more than one “outcome” can be defined using a union type.

```java
File|Path|SyntaxError parsePath(String path) => ... ;
```

We can handle the various outcomes using `switch`:

```java
value result = parsePath(name);
switch (result)
    case (is File) {
        return lines(result);
    }
    case (is Url) {
        return Request(result).execute().content.lines;
    }
    case (is SyntaxError) {
        return {};
    }
```
**Idiom #1**

**Idiom: functions with multiple outcomes**

We can aggregate cases using union:

```java
value result = parsePath(name);
switch (result)
    case (is File|Url) {
        ...
    }
else {
    ...
}
```

Or, alternatively, using `if` instead of `switch`:

```java
if (is File|Url result = parsePath(name)) {
    ...
}
```
Idiom #2

Idiom: functions returning null

Example: retrieve an item from a map.
(Nothing more than a special case of multiple outcomes!)

```
Item? get(Key key) => ... ;
```

Here Item? literally means Nullable<Item>.

```java
value map = HashMap { “CET”->cst, “GMT”->gmt, “PST”->pst };

Timezone tz = map[id]; //not well-typed!
value offset = map[id].rawOffset; //not well-typed!

Timezone? tz = map[id];
value offset = (map[id] else gmt).rawOffset;
```

For a union type of this very common form, we have special syntax sugar.
Idiom #3

Idiom: heterogeneous collections

What is the type of a list containing Integers and Floats?

```java
List<Number> list = Arrays.asList(1, 2, 1.0, 0.0);

Number element = list.get(index);
//handle which the subtypes of Number?
//don't forget that an out of bounds
//index results in an exception
```

The element type is ambiguous, so I must be explicit. Even then I lose some information.
Idiom #3

**Idiom: heterogeneous collections**

With union and intersection, type inference is unambiguous!

```java
value list = ArrayList { 1, 2, 1.0, 0.0 };
```

The inferred element type is `Integer | Float`, resulting in the inferred type `ArrayList<Integer | Float>`, which is a subtype of any type to which the `ArrayList` may be soundly assigned.

No loss of precision!

```java
Integer | Float | Null element = list[index];
//now I know exactly which cases I have to handle
```
Idiom #4

Idiom: unions and streams

Example: the follow() method of Iterable adds an element to the start of a stream.

```
{Element|Other+} follow<Other>(Other element)
    => { element, *this };
```

The syntax `{T*}` and `{T+}` is sugar for the interface Iterable.

Exactly the right type pops out automatically.

```
{String*} words = { “hello”, “world” };
{String?+} strings = words.follow(null);
```

(Even though I’m explicitly writing in the types, I could have let them be inferred.)
Idiom #5

Idiom: intersections and streams

Example: the `coalesce()` function eliminates null elements from a stream.

```cpp
{Element&Object*} coalesce<Element>({Element*} elements)
  => { for (e in elements) if (exists e) e };
```

Again, exactly the right type pops out automatically.

```cpp
{String?*} words = { "hello", null, "world" };
{String*} strings = coalesce(words);
```

(Again, I could have let the types be inferred.)
Idiom #6

Idiom: empty vs nonempty

Problem: the max() function can return null, but only in the case that the stream might be empty. So let's try this:

```cpp
shared Value? max<Value>({Value*} values)
given Value satisfies Comparable<Value> { ... }
```

What if we know it's nonempty at compile time? Do we need a separate function?

```cpp
shared Value maxNonempty<Value>({Value+} values)
given Value satisfies Comparable<Value> { ... }
```

Terrible! This doesn't let us abstract.
Idiom #6

Idiom: empty vs nonempty

Solution: the `Iterable` type has an extra type parameter:

```plaintext
shared Absent|Value max<Value,Absent>(Iterable<Value,Absent> values)  
given Value satisfies Comparable<Value>  
given Absent satisfies Null { ... }
```

Exactly the right type pops out automatically. (And may be inferred.)

```plaintext
Null maxOfNone = max {}; //known to be empty  
String maxOfSome = max { “hello”, “world” }; //known to be nonempty

{String*} noneOrSome = ... ;  
String? max = max(noneOrSome); //might be empty or nonempty
```
Idiom #7

Idiom: multiple return values

For example, an operation might return a Protocol and a Path.

```java
//Java
class ProtocolAndPath { ... }

ProtocolAndPath parseUrl(String url) {
    return new ProtocolAndPath(protocol(url), path(url));
}
```

We have to define a class.
**Idiom #7**

**Idiom: multiple return values**

A function can be defined to return a tuple type.

```java
[Protocol, Path] parseUrl(String url) => [protocol(url), path(url)];
```

Now a caller may extract the individual return values:

```java
value protocolAndPath = parseUrl(url);
Protocol name = protocolAndPath[0];
Path address = protocolAndPath[1];
```

What about other indexes?

```java
Null missing = protocolAndPath[3];
Protocol | Path | Null val = nameAndAddress[index];
```
Idiom #8

**Idiom: spreading tuple return values**

Imagine we want to pass the result of `parseUrl()` to another function

```
Response get(Protocol name, Path address) => ... ;
```

We can use the spread operator, `*`, like in Groovy:

```
value response = get(*parseUrl(url));
```

Or we can work at the function level, using `unflatten()`

```
Response(String) get = compose(unflatten(get), parseUrl);
value response = get("http://ceylon-lang.org");
```

There is a deep relationship between function types and tuple types.
Idiom #9

Idiom: abstract over function types

Problem: the \texttt{compose()} function composes functions.

\begin{align*}
X(A) \text{ compose}<X,Y,A> & (X(Y)x, Y(A)y) \\
& \Rightarrow (A a) \Rightarrow x(y(a));
\end{align*}

But this is not quite as general as it could be!

For functions with just one parameter it works well:

\begin{verbatim}
Anything(Float) printSqrt = compose(print,sqrt);
\end{verbatim}

What about functions with multiple parameters?

\begin{verbatim}
value printSum = compose(print,\texttt{plus});
\end{verbatim}
Idiom #9

Idiom: abstract over function types

Solution: abstract over unknown tuple type.

```
X(*Args) compose<X,Y,Args>(X(Y) x, Y(*Args) y)
given Args satisfies Anything[]
=> flatten((Args args) => x(y(*args)));
```

A little uglier, but does the job!

```
Anything(Float,Float) printSum = compose(print,plus);
```

Even if this doesn't seem that useful at first sight, we actually use it in all sorts of places: for example, in the metamodel, and in ceylon.promise