# Tuple return

A method’s arguments can be considered to be a tuple.

A method’s return values can be considered to be a tuple.

Consider a method foo() with any number of return values (0=void, 1=normal single return, >1 = multi-return). For example:

public void foo();

public Int foo();

public (Int, String) foo();

Regardless, a method can be invoked with a Tuple return. A void return is converted to a Tuple of size 0, a single return to a Tuple of size 1, and so on. The compiler determines that the invocation should produce a Tuple return based on the elicitation of (for example) the L-Value, e.g.:

Tuple t = foo();

Question is how to differentiate:

public Tuple foo();

public (Int, String) foo();

Possibilities:

1. **Preferred**: Disallow those two methods, as they are ambiguous
2. Default to selecting the “Tuple foo()” method if the LValue is a Tuple
3. Provide a means to specify which one is being called

Other things to consider:

* Named return values: Tuple will support optional name for each element, e.g. public (Int count, String name) foo();
* Should the empty tuple be allowed to be used instead of specifying void? e.g. public () foo(); // YUCK!

# Conditional return (conditional method)

Given a declaration such as:

public conditional String next();

The meaning is that the method has a *conditional return*. A conditional return is one that either has a value, or it does not. In the above example, the return type appears to be “String”, but is actually a Conditional Tuple of Boolean and String, where the second (String) value is only available iff the Boolean value is True.

(An attempt to access values when the Boolean value is False will cause a runtime exception, and should cause a compile time warning or error if the likelihood thereof is determinable.)

This is also supported by the if and while statements:

if (String s : next()) {…}

while (String s : next()) {…}

# Future return

A *future return* is associated with calls to methods on a *Service* reference. All methods on a service can be treated as async calls by eliciting a return of a FutureRef.

Consider:

service Example { String foo() {..} }

One can call a service and “block for” the result:

Example example = …

String s = example.foo();

What this is equivalent to is:

Example example = …

@future String sFuture = example.foo();

// implicitly: &sFuture.waitForCompletion();

String s = &sFuture.get();

The interesting thing is that the caller can elicit either a blocking call or an explicit future result. (The compiler must be able to unambiguously determine which.) It is expected that the efficiency of a blocking call will be no lower than (and likely measurably higher than) the efficiency of obtaining an explicitly future result, as the future itself may be optimizable out; however, the behavior must conform to all of the rules laid out for calls to a service, and the behavior or waiting on a future for its result.