# The Ceylon Type System

Lucas Werkmeister

EnthusiastiCon 2018

# Type systems

```
function timesTwo(num) {
  return num * 2;
}
```

# Type systems

```
function timesTwo(Float num) {
  return num * 2;
}
```

# Type systems

```
Float     timesTwo(Float num) {
  return num * 2;
}
```

# Type systems

```
Float     timesTwo(Float num) {
  return num << 1;
}
```

# Type systems

```
Float     timesTwo(Float num) {
  return num << 1; // error: only ints support bitshift
}
```

# Union types

A value of type `X | Y` can have the type `X` or `Y`.

# Union types

A value of type `X|Y` can have the type `X` or `Y`.

```
shared Document|ParseError parseDocument(String html);
```

# Union types

A value of type `X|Y` can have the type `X` or `Y`.

```
shared Document|ParseError parseDocument(String html);
```

```
shared void enqueue(Element item, Integer? priority = null);
```

# Union types

A value of type `X|Y` can have the type `X` or `Y`.

```
shared Document|ParseError parseDocument(String html);
```

```
shared void enqueue(Element item, Integer? priority = null);
```

`Integer?` is equivalent to `Integer|Null`. (`Null` is a class with only one instance, `null`.)

# Intersection types

A value of type `X&Y` has the types `X` and `Y`.

# Intersection types

A value of type `X&Y` has the types `X` and `Y`.

```
shared void printScaled(
    Printable&Scalable element, Float factor);
```

# Flow-sensitive typing

Instead of Java's

```
Object o = getSomething();
if (o instanceof String) {
  String s = (String)o;
  String su = s.toUpperCase();
}
```

# Flow-sensitive typing

Instead of Java's

```
Object o = getSomething();
if (o instanceof String) {
  String s = (String)o;
  String su = s.toUpperCase();
}
```

you simply write

```
Object o = getSomething();
if (is String o) {
  // o has type String now
  String su = o.uppercased; // look ma, no cast!
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) {
  // arg: String



  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: String




  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object




  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object




  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object (String ∪ Null) ∩ Object




  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object
```

$(\texttt{String} \cup \texttt{Null}) \cap \texttt{Object}$

$(\texttt{String} \cap \texttt{Object}) \cup (\texttt{Null} \cap \texttt{Object})$

```
  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object
  //    = <String&Object>|<Null&Object>



  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object
  //    = <String&Object>|<Null&Object>
  //    = String|<Null&Object>


  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object
  //    = <String&Object>|<Null&Object>
  //    = String|<Null&Object>
  //    = String|Nothing

  print(arg.uppercased);
}
```

# Flow-sensitive typing

```
String? arg = process.arguments.first;
if (exists arg) { // is Object arg
  // arg: <String?>&Object
  //    = <String|Null>&Object
  //    = <String&Object>|<Null&Object>
  //    = String|<Null&Object>
  //    = String|Nothing
  //    = String
  print(arg.uppercased);
}
```

# Thank you!

- If this sounded interesting, please talk to me later or keep in touch!
- `@LucasWerkmeistr` or `@ceylonlang`
- `https://ceylon-lang.org`
  - `https://ceylon-lang.org/documentation/current/introduction/`
  - `https://ceylon-lang.org/documentation/current/tour/`

# Backup slide!

More cool stuff!

- Declaration-site variance: `Set<Integer>|Set<Float> = Set<Integer|Float>`
- Tuple types like `[String, Integer, Integer]` are linked lists of types (arbitrary length)
- Tuple types are used for function types like `Integer(Integer, Integer)` (no `Function22` limit like in Scala)