

class: center, middle

# Types

.byline[ Will Billingsley, CC-BY ]

////////////////////////////////////

class: bigquote

## Do these makes sense...

\_\_\_\_\_

\_\_\_\_\_

| 1 plus 1 equals 2

\_\_\_\_\_

| red plus green equals yellow

\_\_\_\_\_

| 1 plus red

////////////////////////////////////

class: bigquote

## What about...

\_\_\_\_\_

\_\_\_\_\_

| red plus rabbit

////////////////////////////////////

class: bigquote

## What about...

\_\_\_\_\_

| red plus rabbit equals a red rabbit

////////////////////////////////////

class: bigquote

## What about...

\_\_\_\_\_

rabbit plus rabbit

class: bigquote

## What about...

rabbit plus rabbit equals lots of baby rabbits

class: bigquote

## Back to numbers

1 plus 1 equals 2

1 apple plus 1 orange

a plus b

class: bigquote, middle

## Let's do something a little engineering

class: bigquote, middle

$$\left( 3 V \times \left( \frac{12 V}{6 \Omega} + 2 A \right) \right)$$

class: bigquote, middle

$$\left( 3 V \times \left( \frac{12 V}{6 \frac{V}{A}} + 2 A \right) \right)$$

class: bigquote, middle

$$\left( 3 V \times \left( \frac{12 (V \times A)}{6 V} + 2 A \right) \right)$$

class: bigquote, middle

$$\left( 3 V \times \left( \frac{12}{6} A + 2 A \right) \right)$$

class: bigquote, middle

$$\lambda (3 V \times (\frac{12}{6} A + 2 A))$$

class: bigquote, middle

$$\lambda (3 V \times (2 A + 2 A))$$

class: bigquote, middle

$$\lambda (3 V \times 4 A)$$

class: bigquote, middle

$$\lambda (3 \times 4 (V \times A))$$

class: bigquote, middle

$$\lambda (12 (V \times A))$$

class: bigquote, middle

$$\lambda (12 W)$$

## The message here...

- Just as we can simplify an equation of values, we can also simplify an equation of types
- Formally, this is called *type theory*. And it is closely related to *automated theorem proving* -- getting a computer to prove the proposition that "this expression is the right type".
- In practice, there is a very close link between programs and proofs -- computer proofs look like programs. *Curry-Howard correspondance*.

## Different type systems

- *When* do we check the types?
    - compile time? "Static typing"
    - run time? "Dynamic typing"
  - *What do we do* if we find the type is incompatible?
    - throw an error? "Strong typing"
    - fudge it somehow? "Weak typing"
- 

## Weak typing in JavaScript

---

What should these resolve to?

```
if (3 ^ "apples" == 4 ^ "oranges") { console.log("Yes, it does!") }
```

```
a == !!a
```

```
17 + ("apples" != "oranges") + "foo"
```

---

## A successful checker

---

- If I'm right, I want my code to run happily...
  - If I'm wrong, I want to know I'm wrong *now*, not *in production*
- 
- Fixing a bug in production costs 10 times as much as before I've released it
- 

## A successful checker

---

- Lets us express every program we want
  - Stops us from expressing every program we obviously *didn't* want
- 
- But isn't this verbose?

```
Map<Student, List<Course>> courseMap = new HashMap<Student, List<Course>>();
```

```
courseMap = {}
```

---

## Type inference

---

- Let's keep the benefits of static typing (finding errors sooner, tool support, eg, auto-completion in IDEs)
- But lose some of the verbosity.
- What type are these?

---

```
val a = 127
```

---

```
val b = "Hello world"
```

---

```
val b = List("one", "two", "three").head
```

---

- Type inference -- so easy, you can do it already!
- 

---

## Type inference gotchas

---

- Sometimes it needs some help

```
def factorialStep(soFar:Long, thisNum:Int) = {  
  if (thisNum == 1) {  
    soFar  
  } else {  
    factorialStep(thisNum * soFar, thisNum - 1)  
  }  
}
```

```
[error] /Users/wbilling/sourcecode/teaching/cosc250/firstscala/src/main/scala/
cosc250/firststeps/StepOne.scala:63: recursive method factorialStep needs resu
lt type
[error]           factorialStep(thisNum * soFar, thisNum - 1)
[error]           ^
```

---

## Type inference gotchas

- Sometimes it needs some help

```
def factorialStep(soFar:Long, thisNum:Int):Long = {
  if (thisNum == 1) {
    soFar
  } else {
    factorialStep(thisNum * soFar, thisNum - 1)
  }
}
```

---

## Type inference gotchas

- Sometimes it can be too narrow

```
trait ContentItem {
  def copyrightHolder = None
}

class Book(val title:String, val author:String) extends ContentItem {
  override def copyrightHolder = author
}
```

```
Expression of type Some[String] does not conform to expected type None.type
```

---

## Inheritance and types

- In one JAR we might have

```
public interface HasId {
  String getId();
}
```

and in another JAR entirely we might have

```
public class Foo extends HasId {  
  
    String id;  
  
    @Override  
    public String getId() {  
        return this.id;  
    }  
}
```

---

## But what about this

```
public class User {  
    public String getId() {  
        return this.id;  
    }  
}
```

- 
- Subtypes represent an "is a" relationship

- 
- Is it a `HasId` ?

- 
- Java uses *Nominal subtyping* for inheritance -- classes declare the contracts they conform to
- 

## Structural subtyping

---

```
object Demo {  
  
  type HasId = {  
    def getId:String  
  }  
  
  class Book(val title:String, val author:String) {  
    def getId = ""  
  }  
  
  val a: HasId = new Book("a", "b")  
}
```

---

## Ad-hoc polymorphism (typeclasses)

- Common in Haskell, fairly common in Scala.

Are these *Openable*?

- A can
- A beer bottle
- A wine bottle (with a cork)

---

## Ad-hoc polymorphism (typeclasses)

- Common in Haskell, fairly common in Scala.

Are these *Openable*?

- A can -- **only if I've got a can opener!**
- A beer bottle -- **only if I've got a bottle opener!**
- A wine bottle -- **only if I've got a corkscrew!**

---

## Ad-hoc subtyping (typeclasses)

- A trait for the **evidence**

```
trait Openable[A] {  
  def open(a:A): String  
}
```



- 
- A wine bottle

```
class Wine(val name:String)
```

---

## Ad-hoc subtyping (typeclasses)

---

- A function, needing something openable

```
def pour[A](item: A)(implicit ev: Openable[A]) = {  
  val contents = ev.open(item)  
  println(contents)  
  contents  
}
```

- 
- Let's call it

```
val w = new Wine("Beaujolais") // we need a second parameter!
```

- 
- Let's provide the corkscrew

```
public object Wine {  
  implicit object Corkscrew extends Openable[Wine] {  
    def open(w:Wine) = "Lovely " + w.name  
  }  
}
```

---

## Syntactic sugar

---

- A function, needing something openable

```
def pour[A](item: A)(implicit ev: Openable[A]) = {  
  val contents = ev.open(item)  
  println(contents)  
  contents  
}
```

---

```
def pour[A : Openable](item: A) = {  
  val opener = implicitly[Openable[A]]  
  val contents = opener.open(item)  
  println(contents)  
  contents  
}
```

---

## Parametrized Types

- In Java, called *Generics*
- In Scala, you just saw them
  - `List[String]`
  - `Openable[A]`
  - etc

---

## Parametrized Types and Subtyping

- If a `Student` is a `Person`, is a `List[Student]` a `List[Person]` ?

```
import scala.collection.mutable  
  
trait Person  
object Bob extends Person  
class Student(n:String) extends Person  
  
def addBob(seq:mutable.Buffer[Person]) = {  
  seq.append(Bob)  
}  
  
val a = mutable.Buffer(new Student("Fred"), new Student("Alice"))  
  
addBob(a)
```

---

## Covariance and Contravariance

- `class Container[+A]`

if `Y` is a subtype of `X`, `Container[Y]` is a subtype of `Container[X]`

- `class Container[-A]`

if `Y` is a subtype of `X`, `Container[X]` is a subtype of `Container[Y]`

- `class Container[A]`

`Container[Y]` and `Container[X]` are unrelated

---

- This is fairly common:

```
class Container[-A, +B]
```

---

## A bit of fun with implicits

---

- Scala can do *implicit conversions* that let you decorate classes
  - eg, locally make it seem like you've added methods to someone else's library

- It also has a convention where

```
obj.callMyFunc(param)
```

can also be written

```
obj callMyFunc param
```

- Let's make `"one" plus "one" equals "two"`