Academese to English:
A Practical Tour of Scala's Type System

Heather Miller
@heathercmiller

PhillyETE, April 11th, 2016

scalacenter

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Motivation for this talk:

SCALA'S GOT A VERY RICH TYPE SYSTEM

You can do a ton of stuff with it.

Yet, the basics could really be better explained.

RULE:

Let's only look at stuff that 80% of people can rapidly apply.

Who is this talk for?

Everyone.
...except Scala type system experts.

MY GOAL:

To show you some of the basics of Scala's type system. Just the handful of concepts you should know to be proficient.

Nothing fancy.

Topics we'll cover:

- Scala's basic pre-defined types
- Defining your own types
- Parameterized types
- Bounds
- Variance
- Abstract types
- Existential types
- Type classes

There's a list of other stuff this talk won't cover.

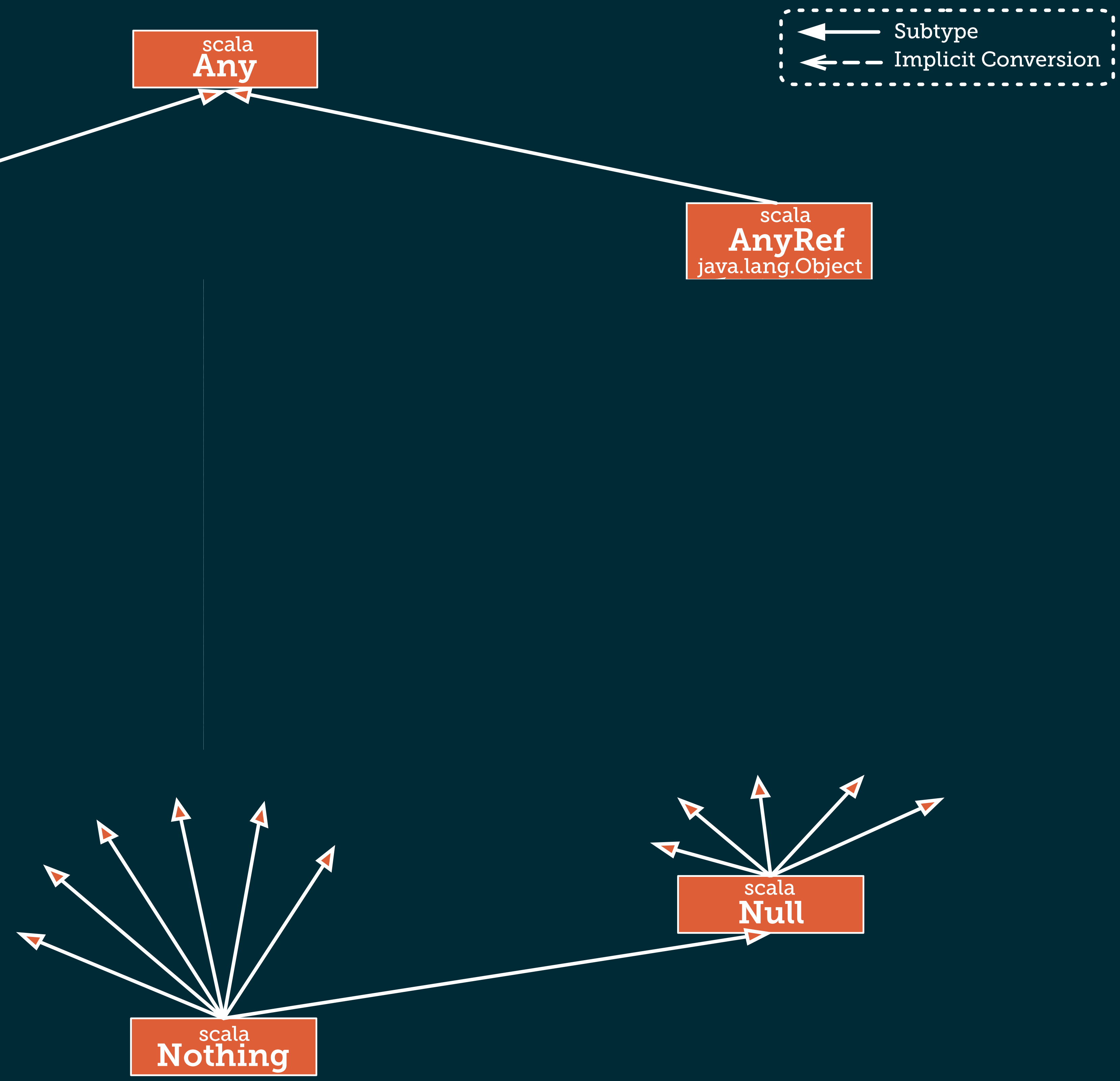
e.g., Type-level programming, Higher Kinded Types, Path-Dependent Types, ..., Dotty.

Let’s go on....

A Whirlwind tour of Scala’s type system

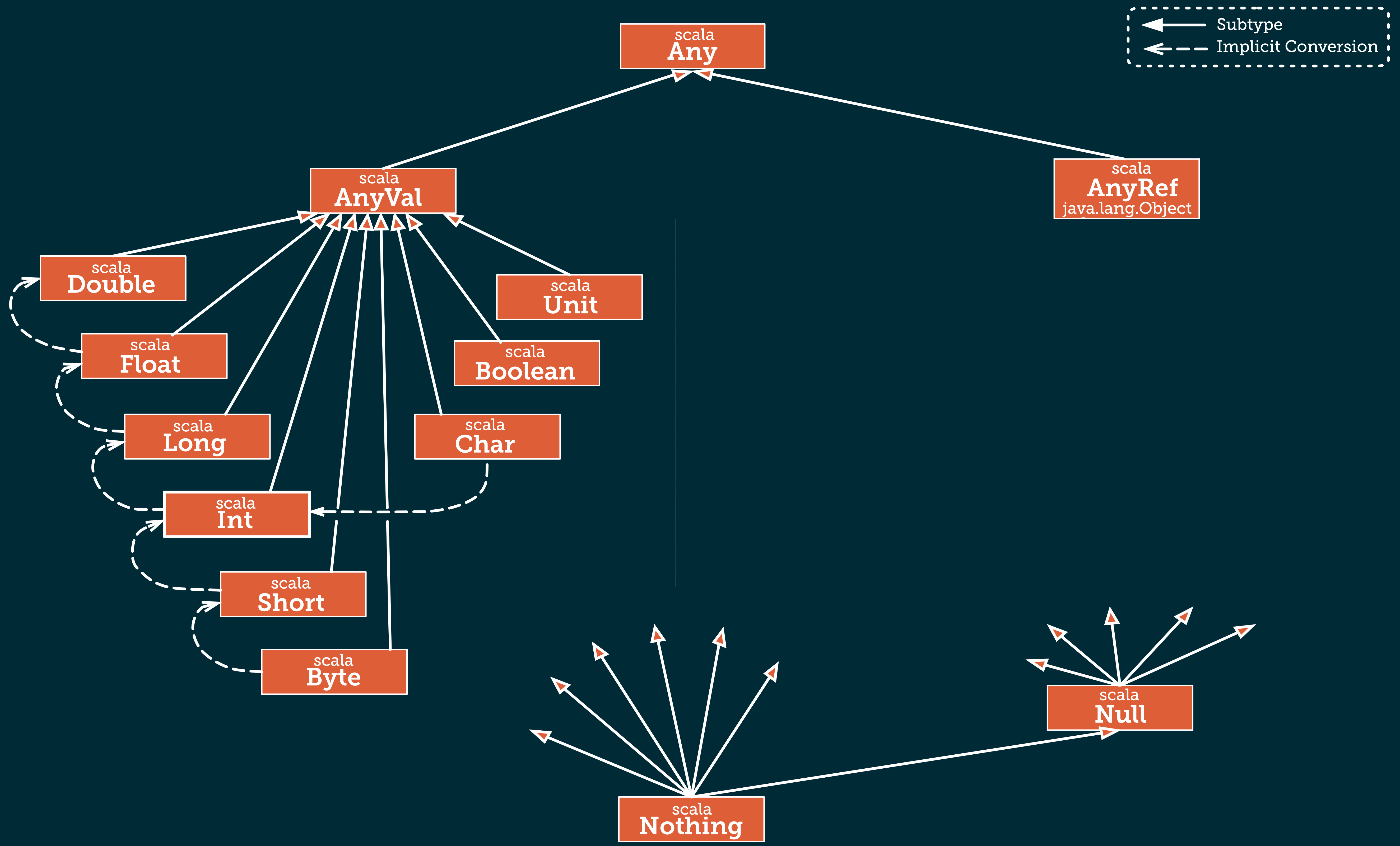
Scala's

Basic predefined types



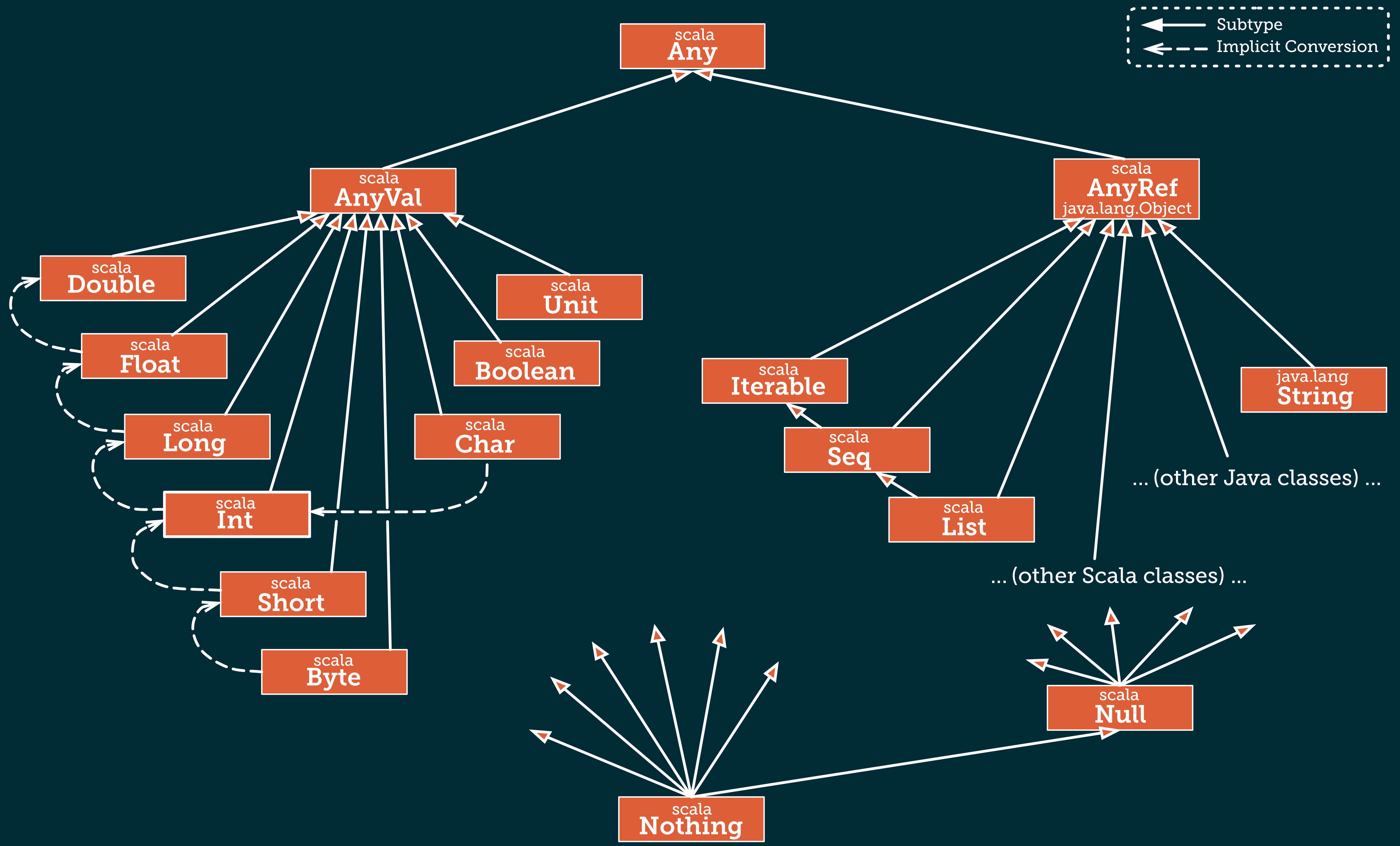
Scala's

Basic predefined types



Scala's

Basic predefined types



How do we

Define our own types?

TWO WAYS:

1.) **Declarations** of named types
e.g., traits or classes

- **Define** a class or a trait
- **Define** a type member using the type keyword

```
class Animal(age: Int) {
  // fields and methods here...
}

trait Collection {
  type T
}
```

How do we
Define our own types?

TWO WAYS:

1.) **Declarations** of named types
e.g., traits or classes

→ **Define** a class or a trait
→ **Define** a type member using the type keyword

2.) **Combine**. Express types (not named) by combining existing types.

→ *e.g., compound type, refined type*

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  //...
}
```

Interacting with typechecking via

Parameterized Types

WHAT ARE THEY?

Same as generic types in Java. A generic type is a generic class or interface that is parameterized over types.

for example:

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

Interacting with typechecking via

Parameterized Types

WHAT ARE THEY?

Same as generic types in Java. A generic type is a generic class or interface that is parameterized over types.

for example:

```
class Stack[T] {
  var elems: List[T] = Nil
  def push(x: T) { elems = x :: elems }
  def top: T = elems.head
  def pop() { elems = elems.tail }
}
```

Can interact with type-checking by adding or relaxing constraints on the type parameters

using

VARIANCE

BOUNDS

Parameterized types; you can constrain them.

Bounds?

Both type parameters and type members can have type bounds:

- lower bounds (subtype bounds)
- upper bounds (supertype restrictions)

for example:

```
trait Box[T <: Tool]
```

```
trait Generic[T >: Null] {
  // `null` allowed due to lower
  //  bound
  private var fld: T = null
}
```

Remember the type hierarchy?

All types have an upper bound of `Any` and a lower bound of `Nothing`

Parameterized types; you can constrain them.

Bounds?

Both type parameters and type members can have type bounds:

- lower bounds (subtype bounds)
- upper bounds (supertype restrictions)

for example:

```
trait Box[T <: Tool]
```

A **Box** can contain any element **T** which is a subtype of `Tool`.

Remember the type hierarchy?

All types have an upper bound of `Any` and a lower bound of `Nothing`

Parameterized types; you can constrain them.

Bounds?

Both type parameters and type members can have type bounds:

- lower bounds (subtype bounds)
- upper bounds (supertype restrictions)

Null can be used as a bottom type for any value that is nullable.

for example:

```
trait Box[T <: Tool]
```

```
trait Generic[T >: Null] {
  // `null` allowed due to lower
  //  bound
  private var fld: T = null
}
```

Remember the type hierarchy?

All types have an upper bound of Any and a lower bound of Nothing

Parameterized types; you can constrain them.

Bounds?

Both type parameters and type members can have type bounds:

- lower bounds (subtype bounds)

Null can be used as a bottom type for any value that is nullable.

for example:

Recall class Null from the type hierarchy. It is the type of the null reference; it is a subclass of every reference class (i.e., every class that itself inherits from AnyRef).

Null is not compatible with value types.

```
trait Generic[T >: Null] {
  // `null` allowed due to lower
  //  bound
  private var fld: T = null
}
```

```
scala> val i: Int = null
<console>:4: error: type mismatch;
 found   : Null(null)
 required: Int
```

Parameterized types; you can constrain them.

Variance?

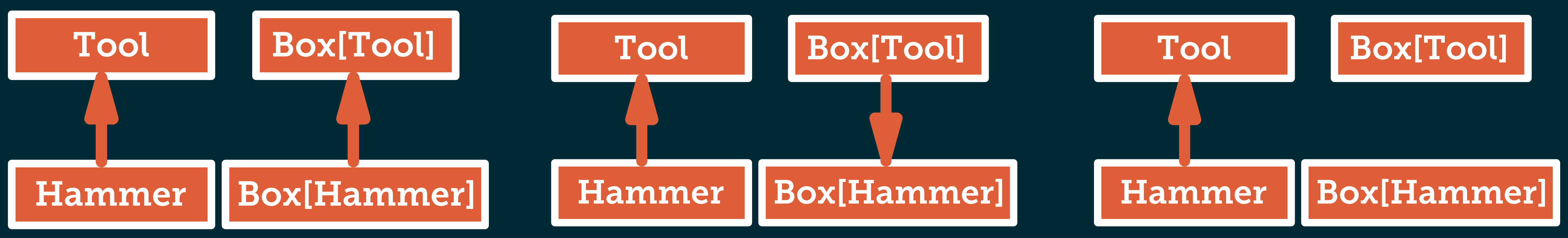
Given the following:

```
trait Box[T]
class Tool
class Hammer extends Tool
```

How might they relate to one another?

THREE POSSIBILITIES:

Covariant **Contravariant** **Invariant**



Covariance

Let's look at a simple zoo-inspired example. Given:

```
trait Animal
class Mammal extends Animal
class Zebra extends Mammal
```

We'd like to define a field for our animals to live on:

```
abstract class Field[A] {
  def get: A
}
```

Now, let's define a function **`isLargeEnough`** that takes a **`Field[Mammal]`** and tests if the field is large enough for the mammal to live in

```
def isLargeEnough(run: Field[Mammal]): Boolean = …
```

Can we pass zebras to this function? A Zebra is a Mammal, right?

http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/

Covariance

```
scala> isLargeEnough(zebraRun)
<console>:14: error: type mismatch;
 found   : Run[Zebra]
 required: Run[Mammal]
```

Nope. Field[Zebra] is not a subtype of Field[Mammal]. Why? Field, as defined is invariant. There is no relationship between Field[Zebra] and Field[Mammal].

So let's make it covariant!

```
abstract class Field[+A] {
  def run: A
}
```

Et voilà, it compiles.

http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/

Contravariance

Keeping with our zoo-inspired example, let's say our zoo has several vets. Some specialized for specific species.

```
abstract class Vet[A] {
  def treat(a: A)
}
```

We need just one vet to treat all the mammals of our zoo:

```
def treatMammals(vet: Vet[Mammal]) { … }
```

Can we pass a vet of animals to `treatMammals`?

A Mammal is an Animal, so if you have a vet that can treat animals, it will be OK to pass a mammal, right?

http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/

Contravariance

```
scala> treatMammals(animalVet)
<console>:14: error: type mismatch;
 found   : Vet[Animal]
 required: Vet[Mammal]
```

Nope. This doesn't work because Vet[Animal] is not a subtype of Vet[Mammal], despite Mammal being a subtype of Animal.

We want Vet[A] to be a subtype of Vet[B] if B is a subtype of A.

So let's make it contravariant!

```
abstract class Vet[-A] {
  def treat(a: A)
}
```

Et voilà, it compiles.

http://julien.richard-foy.fr/blog/2013/02/21/be-friend-with-covariance-and-contravariance/

Wait, what's the difference between A<:B and +B?

They seem kind of similar, right?
They're different!

`Coll[A<:B]` means that class `Coll` can take any class `A` that is a subclass of `B`.

`Coll[+B]` means that `Coll` can take any class, but if `A` is a subclass of `B`, then `Coll[A]` is considered to be a subclass of `Coll[B]`.

http://stackoverflow.com/questions/4531455/whats-the-difference-between-ab-and-b-in-scala

Wait, what's the difference between A<:B and +B?

They seem kind of similar, right?
They're different!

`Coll[A<:B]` means that class `Coll` can take any class `A` that is a subclass of `B`.

Useful when you want to be generic but require a certain set of methods in B

`Coll[+B]` means that `Coll` can take any class, but if `A` is a subclass of `B`, then `Coll[A]` is considered to be a subclass of `Coll[B]`.

Useful when you want to make collections that behave the same way as the original classes

http://stackoverflow.com/questions/4531455/whats-the-difference-between-ab-and-b-in-scala

Wait, what's the difference between A<:B and +B?

Said another way... Given:

```
class Animal
class Dog extends Animal

class Car
class SportsCar extends Car
```

VARIANCE:

```
case class List[+B](elements: B*) {} // simplification

 val animals: List[Animal] = List( new Dog(), new Animal() )
 val cars: List[Car] = List ( new Car(), new SportsCar() )
```

As you can see **List does not care whether it contains Animals or Cars**. The developers of List did not enforce that e.g. only Cars can go inside Lists.

http://stackoverflow.com/questions/4531455/whats-the-difference-between-ab-and-b-in-scala

Wait, what's the difference between A<:B and +B?

Said another way... Given:

```scala
class Animal
class Dog extends Animal

class Car
class SportsCar extends Car
```

BOUNDS:

```scala
case class Barn[A <: Animal](animals: A*) {}

val animalBarn: Barn[Animal] = Barn( new Dog(), new Animal() )
val carBarn = Barn( new SportsCar() )
// error: inferred type arguments [SportsCar] do not conform to method
// apply's type parameter bounds [A <: Animal]
//     val carBarn = Barn(new SportsCar())
                  ^
```

As you can see **Barn is a collection only intended for Animals**. No cars allowed in here.

http://stackoverflow.com/questions/4531455/whats-the-difference-between-ab-and-b-in-scala

If you're a Java developer, this may not be surprising.

A lot of these things exist for Java.

SO HOW IS THIS RICHER?

Let's look at some other aspects of Scala's type system!

Abstract type members

Basic idea:

A type member (member of an object or class) that is left *abstract*.

Why is this desirable?

Turns out that this is a powerful method of abstraction.

Using abstract type members, we can do a lot of what parameterization does, but is often more flexible/ elegant!

FUNDAMENTAL IDEA:

Define a type and leave it "abstract" until you know what type it will be when you need to make it concrete in a subclass.

Abstract type members

FUNDAMENTAL IDEA:

Define a type and leave it "abstract" until you know what type it will be when you need to make it concrete in a subclass.

Example:

Given:

```
trait Pet
class Cat extends Pet
```

Let's create a person, Susan, who has a Cat both using abstract type members and parameterization.

Abstract type members

FUNDAMENTAL IDEA:

Define a type and leave it "abstract" until you know what type it will be when you need to make it concrete in a subclass.

Example:

Given:
```
trait Pet
class Cat extends Pet
```

```
class Person {
  type Pet
}
class Susan extends Person {
  type Pet = Cat
}
```

Abstract type members

```
class Person[Pet]
class Susan
  extends Person[Cat]
```

Parameterization

Abstract type members

A bigger example from ScalaTest:

Parameterization

```
trait FixtureSuite[F] {
  // ...
}
trait StringBuilderFixture { this: FixtureSuite[StringBuilder] =>
  // ...
}
class MySuite extends FixtureSuite[StringBuilder] with StringBuilderFixture {
  // ...
}
```

Abstract type members

```
trait FixtureSuite {
  type F
  // ...
}
trait StringBuilderFixture { this: FixtureSuite =>
  type F = StringBuilder
  // ...
}
class MySuite extends FixtureSuite with StringBuilderFixture {
  // ...
}
```

http://www.artima.com/weblogs/viewpost.jsp?thread=270195

Abstract type members

A bigger example from ScalaTest:

Parameterization

```
trait FixtureSuite[F] {
  // ...
}
trait StringBuilderFixture { this: FixtureSuite[StringBuilder] =>
  // ...
}
class MySuite extends FixtureSuite[StringBuilder] with StringBuilderFixture {
  // ...
}
```

Abstract type members

```
trait FixtureSuite {
  type F
  // ...
}
trait StringBuilderFixture { this: FixtureSuite =>
  type F = StringBuilder
  // ...
}
class MySuite extends FixtureSuite with StringBuilderFixture {
  // ...
}
```

THE TAKE AWAY:

http://www.artima.com/weblogs/viewpost.jsp?thread=270195

Abstract type members

A bigger example from ScalaTest:

Parameterization

```
trait FixtureSuite[F] {
  // ...
}
trait StringBuilderFixture { this: FixtureSuite[StringBuilder] =>
  // ...
}
class MySuite extends FixtureSuite[StringBuilder] with StringBuilderFixture {
  // ...
}
```

Abstract type members

```
trait FixtureSuite {
```

THE TAKE AWAY:

Abstraction without the verbosity of type parameters. (Can be DRYer).

```
  // ...
}
```

http://www.artima.com/weblogs/viewpost.jsp?thread=270195

Existential types

Basic idea:

Intuitively, an existential type is a type with some unknown parts in it.

```
Wombit[T] forSome { type T }
```

Importantly,

An existential type includes references to abstract type/value members that we know exist, but whose concrete types/values we don't know.

For example, in the above, **T** is a type we don't know concretely, but that we know exists.

Existential types

FUNDAMENTAL IDEA:

Can leave some parts of your program unknown, and still typecheck it with **different implementations** for those unknown parts.

Importantly,

An existential type includes references to abstract type/value members that we know exist, but whose concrete types/values we don't know.

For example, in the above, **T** is a type we don't know concretely, but that we know exists.

Existential types

FUNDAMENTAL IDEA:

Can leave some parts of your program unknown, and still typecheck it with **different implementations** for those unknown parts.

Example:

```
case class Fruit[T](val weight: Int, val tooRipe: T => Boolean)

class Farm {
  val fruit = new ArrayBuffer[Fruit[T] forSome { type T }]
}
```

Note that existentials are safe, whereas Java's raw types are not.

Existential types

Let's look at another example.

```
scala> def foo(x: Array[Any]) = println(x.length)
foo: (Array[Any])Unit

scala> foo(Array("foo", "bar", "baz"))
```

http://www.drmaciver.com/2008/03/existential-types-in-scala/

Existential types

Let's look at another example.

```
scala> def foo(x: Array[Any]) = println(x.length)
foo: (Array[Any])Unit

scala> foo(Array("foo", "bar", "baz"))
:6: error: type mismatch;
 found   : Array[String]
 required: Array[Any]
       foo(Array[String]("foo", "bar", "baz"))
```

This doesn't compile, because an Array[String] is not an Array[Any].

However, it's completely typesafe—we've only used methods that would work for any Array.

How do we fix this?

http://www.drmaciver.com/2008/03/existential-types-in-scala/

Existential types

Attempt #2: Type parameters

```
scala> def foo[T](x: Array[T]) = println(x.length)
foo: [T](Array[T])Unit

scala> foo(Array("foo", "bar", "baz"))
3
```

Now foo is parameterized to accept any T. But now we have to carry around this type parameter, and we know we only care about methods on Array and not what the Array contains. So it's really not necessary.

We can use existentials to get around this.

http://www.drmaciver.com/2008/03/existential-types-in-scala/

Existential types

Attempt #3: Existentials

```
scala> def foo(x: Array[T] forSome { type T}) = println(x.length)
foo: (Array[T] forSome { type T })Unit

scala> foo(Array("foo", "bar", "baz"))
3
```

Woohoo!
Note that a commonly-used shorthand is: Array[_]

> Existential types provide a way of abstracting type information, such that (a) a provider can hide a concrete type ("pack"), and thus avoid any possibility of the client depending on it, and (b) a client can manipulate said type by only by giving it a name ("unpack") and making use of its bounds.
>
> Existentials play a big role in our understanding of abstract data types and encapsulation. - Burak Emir

http://www.drmaciver.com/2008/03/existential-types-in-scala/

Existential types

Attempt #3: Existentials

```
scala> def foo(x: Array[T] forSome { type T}) = println(x.length)
foo: (Array[T] forSome { type T })Unit

scala> foo(Array("foo", "bar", "baz"))
3
```

Woohoo!
Note that a commonly-used shorthand is: Array[_]

> Existential types provide a way of abstracting type information, such that (a) a provider can hide a concrete type ("pack"), and thus avoid any possibility of the client depending on it, and (b) a client can manipulate said type by only by giving it a name ("unpack") and making use of its bounds.
>
> Existentials play a big role in our understanding of abstract data types and encapsulation. - Burak Emir

THE TAKE AWAY:

http://www.drmaciver.com/2008/03/existential-types-in-scala/

Existential types

Attempt #3: Existentials

```
scala> def foo(x: Array[T] forSome { type T}) = println(x.length)
foo: (Array[T] forSome { type T })Unit

scala> foo(Array("foo", "bar", "baz"))
3
```

Woohoo!
Note that a commonly-used shorthand is: Array[_]

> Existential types provide a way of abstracting type information,
> ...
> types and encapsulation. - Burak Emir

http://www.drmaciver.com/2008/03/existential-types-in-scala/

THE TAKE AWAY:

Code reuse: fully decouple implementation details from types

Patterns:

Type classes

Type classes enable retroactive extension.

the ability to extend existing software modules with new functionality without needing to touch or re-compile the original source.

(ad-hoc polymorphism)

Type classes?

Interface: *the "type class"*

```
trait Pickler[T] {
  def pickle(obj: T): Array[Byte]
}
```

Implementation: *the "type class instance"*

```
implicit object intPickler extends Pickler[Int] {
  def pickle(obj: Int): Array[Byte] = {
    // logic for converting Int to Array[Byte]
  }
}
```

Type classes?

Interface:

```
trait Pickler[T] {
  def pickle(obj: T): Array[Byte]
}
```

Implementation:

```
implicit object intPickler extends Pickler[Int] {
  def pickle(obj: Int): Array[Byte] = {
    // logic for converting Int to Array[Byte]
  }
}
```

Type classes?

Interface:

```
trait Pickler[T] {
  def pickle(obj: T): Array[Byte]
}
```

1. The first part is an interface containing one or more operations that should be provided by several different types.

Type classes?

Interface:

```
trait Pickler[T] {
  def pickle(obj: T): Array[Byte]
}
```

1. The first part is an interface containing one or more operations that should be provided by several different types.

Here, a pickle method should be provided for an arbitrary type, T.

Type classes?

2. Implement that interface for different types.

Crucial: the correct implementation must be selected *automatically* based on type!

Implementation:

```
object intPickler extends Pickler[Int] {
  def pickle(obj: Int): Array[Byte] = {
    // logic for converting Int to Array[Byte]
  }
}
```

Type classes?

2. Implement that interface for different types.

Crucial: the correct implementation must be selected *automatically* based on type!

Implementation:

```
implicit object intPickler extends Pickler[Int] {
  def pickle(obj: Int): Array[Byte] = {
    // logic for converting Int to Array[Byte]
  }
}
```

Type classes?

Interface:

```
trait Pickler[T] {
  def pickle(obj: T): Array[Byte]
}
```

Implementation:

```
implicit object intPickler extends Pickler[Int] {
  def pickle(obj: Int): Array[Byte] = {
    // logic for converting Int to Array[Byte]
  }
}
```

Using type classes?

Example user code:

```
def persist[T](obj: T)(implicit p: Pickler[T]): Unit = {
  val arr = obj.pickle
  // persist byte array `arr`
}
```

Type classes automate the selection of the implementation.

Automatic selection is enabled by marking the pickler parameter as implicit!

Using type classes?

Shorthand with context bound!

Example user code:

```
def persist[T: Pickler](obj: T): Unit = {
  val arr = obj.pickle
  // persist byte array `arr`
}
```

Type classes automate the selection of the implementation.

Using type classes?

Example user code:

```
def persist[T](obj: T)(implicit p: Pickler[T]): Unit = {
  val arr = p.pickle(obj)
  // persist byte array `arr`
}
```

Type classes automate the selection of the implementation.

Now possible to invoke persist without passing a pickler implementation explicitly:

```
persist(15)
```

The type checker automatically infers the missing argument to be intPickler, purely based on its type.

Patterns:

Type classes

Example user code:

```
def persist[T](obj: T)(implicit p: Pickler[T]): Unit = {
  val arr = p.pickle(obj)
  // persist byte array `arr`
}
```

Type classes automate the selection of the implementation.

THE TAKE AWAY:

Now possible to invoke persist without passing a pickler implementation explicitly:

```
persist(15)
```

The type checker automatically infers the missing argument to be intPickler, purely based on its type.

Patterns:

Type classes

Example user code:

```
def persist[T](obj: T)(implicit p: Pickler[T]): Unit = {
  val arr = p.pickle(obj)
  // persist byte array `arr`
}
```

Type classes automate the selection of the implementation

No
pi

THE TAKE AWAY:

Retroactively add functionality without having to recompile.

The type checker automatically infers the missing argument to be intPickler, purely based on its type.

That's about all I'll cover.

But there's more.

In addition there's a bunch more one can do:

- Type-level programming.
- Type-based materialization with macros.
- Tricks with path-dependent types.
- Higher-kinded types.

If you're interested, go forth, have fun!

That stuff is advanced. It's not required knowledge to be a good Scala programmer.

You can always do lots of powerful stuff with type parameters/type members, bounds, variance, and type classes – all introduced here!

That's about all I'll cover.

Resources for more advanced stuff

- The Typelevel folks have an amazing blog!
  http://typelevel.org/blog/
- Konrad Malawski has a wiki of type system constructs and patterns
  http://ktoso.github.io/scala-types-of-types/

Thank you!