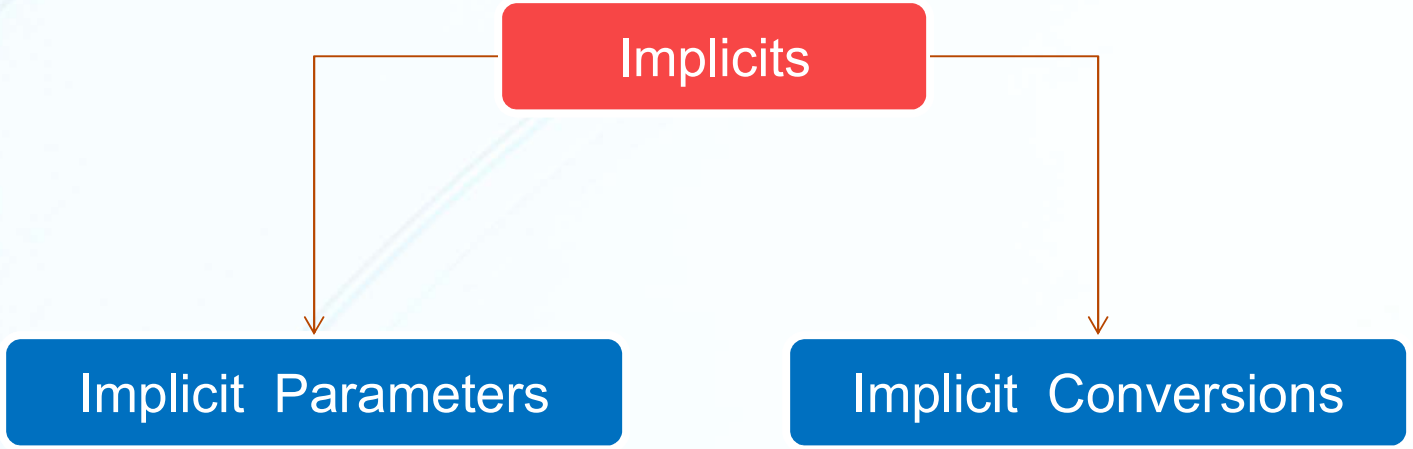
Implicits & Type Classes

IMPLICITS & TYPE CLASSES

Implicits

Implicits



Implicit Parameters

Implicit Parameter is a value that can be passed “automatically”

- A method can have an implicit parameter list, marked by the `implicit` keyword at the start of the parameter list.
- If the parameters in that parameter list are not passed as usual, Scala will look if it can get an implicit value of the correct type, and if it can, pass it automatically.

Implicit Parameters

- An implicit parameter list (implicit *p1,…,pn*) of a method marks the parameters *p1,…,pn* as implicit.
- A method or constructor can have only one implicit parameter list, and it must be the last parameter list given.
- A method with implicit parameters can be applied to arguments just like a normal method. In this case the implicit label has no effect. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided.

Implicit Parameters

Scala looks for implicit parameters in:

- implicit definitions and implicit parameters that can be accessed directly (without a prefix) at the point the method with the implicit parameter block is called.

- Then it looks for members marked implicit in all the companion objects associated with the implicit candidate type.

Implicit Conversions

Implicit Conversions refers to the automatic conversion of one type to another implicitly.

If one calls a method ***m*** on an object ***o*** of a class ***C***, and that class does not support method ***m***, then Scala will look for an implicit conversion from ***C*** to something that does support ***m***.

A simple example would be the method **`map`** on **`String`**:

```
"hello world".map(_.toInt)
```

`String` does not support the method **`map`**, but **`StringOps`** does, and there's an implicit conversion from **`String`** to **`StringOps`** available.

Implicit Classes

Implicit Classes allow us to define all out implicit methods in a single place.

Note, that there are requirements for the class to be implicit:

- It has to be inside another trait, class or object
- It has to have exactly one parameter (but it can have multiple implicit parameters on its own)
- There may not be any method, member or object in scope with the same name

Implicits - Rules

- Implicit definitions are those that the compiler is allowed to insert into a program in order to fix any of its type errors.

- For example, if x + y does not type check, then the compiler might change it to `convert`(x) + y, where `convert` is some available implicit conversion. If `convert` changes x into something that has a + method, then this change might fix a program so that it type checks and runs correctly.

Implicits Rules - Marking

Marking Rule: Only definitions marked `implicit` are available.

- The `implicit` keyword is used, to mark which declarations the compiler may use as implicits.
- You can use it to mark any variable, function, or object definition.

Implicits Rules - Scope

An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion.

- The Scala compiler will only consider implicit conversions that are in scope.
- To make an implicit conversion available, therefore, you must in some way bring it into scope.

When to use implicits?

- Implicits can make code confusing if they are used too frequently. Thus, before adding a new implicit conversion, first ask whether you can achieve a similar effect through other means, such as inheritance, mixin composition, or method overloading.

- If all of these fail, however, and you feel like a lot of your code is still tedious and redundant, then implicits might just be able to help you out.

Type Classes

Type Parameter

- **`List[T] => T`** is called type-parameter;
  `List[T]` is called a generic

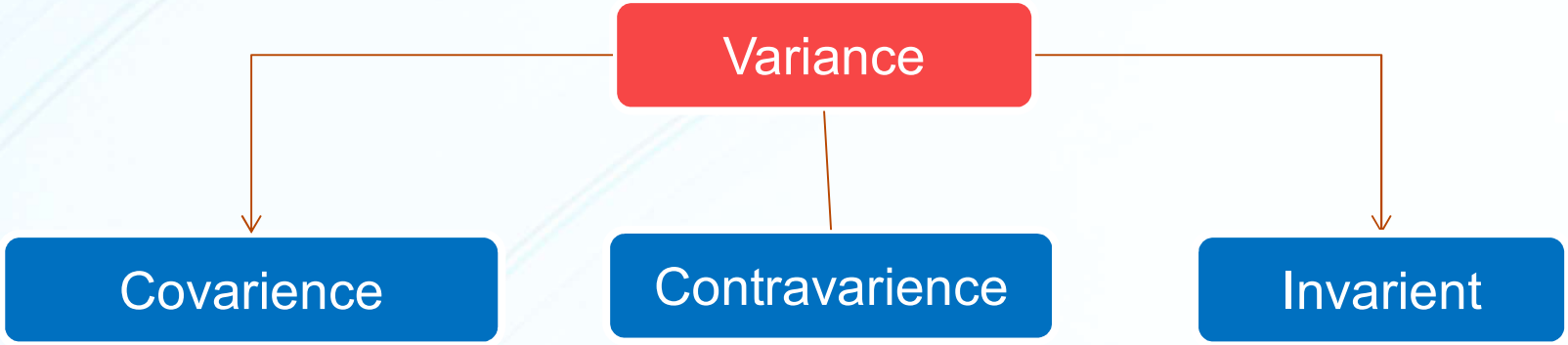
- For **`List[T]`**, if we use **`List[Int]`**,**`List[AnyVal]`** etc. then these **`List[Int]`** and **`List[AnyVal]`** are known as "**Parameterized Types**".

- Variance defines Inheritance relationship between these Parameterized Types.

Variance

- Variance defines Inheritance relationships of Parameterized Types. Variance is all about Sub-Typing.
- A type parameter of a class or trait can be marked with a variance annotation, either *covariant* (+) or *contravariant* (). Such variance annotations indicate how sub-typing works for a generic class or trait.
- For example, the generic class List is covariant in its type parameter, and thus List[String] is a subtype of List[Any].

Variance



- Covariant [+T] → If S is subtype of T, then List[S] is also subtype of List[T]
- Contravariant [-T] → If S is subtype of T, then List[T] is also subtype of List[S]
- Invariant[T] → If S is subtype of T, then List[S] and List[T] are unrelated.

Type Classes

- The combination of **parameterized types** and **implicit parameters** is called **type classes**.

- Type classes are a powerful and flexible concept that adds ad-hoc polymorphism to Scala.

Type Classes

- Type class is a class (group) of types, which satisfy some contract defined in a trait with addition that such functionality (trait and implementation) can be added without any changes to the original code.

- One could say that the same could be achieved by extending a simple trait, but with type classes it is not necessary to predict such a need beforehand.

THANK YOU