Variance
Variance

You have seen that some types should be covariant whereas others should not.

Roughly speaking, a type that accepts mutations of its elements should not be covariant.

But immutable types can be covariant, if some conditions on methods are met.
Definition of Variance

Say $C[T]$ is a parameterized type and $A, B$ are types such that $A <: B$.

In general, there are *three* possible relationships between $C[A]$ and $C[B]$:

- $C[A] <: C[B]$  
  - C is *covariant*
- $C[A] >: C[B]$  
  - C is *contravariant*
- neither $C[A]$ nor $C[B]$ is a subtype of the other  
  - C is *nonvariant*
Definition of Variance

Say \( C[T] \) is a parameterized type and \( A, B \) are types such that \( A <: B \).

In general, there are *three* possible relationships between \( C[A] \) and \( C[B] \):

- \( C[A] <: C[B] \) \( \quad \text{C is covariant} \)
- \( C[A] >: C[B] \) \( \quad \text{C is contravariant} \)
- neither \( C[A] \) nor \( C[B] \) is a subtype of the other \( \quad \text{C is nonvariant} \)

Scala lets you declare the variance of a type by annotating the type parameter:

- `class C[+A] { ... }` \( \quad \text{C is covariant} \)
- `class C[-A] { ... }` \( \quad \text{C is contravariant} \)
- `class C[A] { ... }` \( \quad \text{C is nonvariant} \)
Exercise

Say you have two function types:

```haskell
  type A = IntSet => NonEmpty
  type B = NonEmpty => IntSet
```

According to the Liskov Substitution Principle, which of the following should be true?

0   A <: B
0   B <: A
0   A and B are unrelated.
Exercise

Say you have two function types:

\[
\text{type } A = \text{IntSet} \Rightarrow \text{NonEmpty} \\
\text{type } B = \text{NonEmpty} \Rightarrow \text{IntSet}
\]

According to the Liskov Substitution Principle, which of the following should be true?

- A :: B
- B :: A
- A and B are unrelated.
Typing Rules for Functions

Generally, we have the following rule for subtyping between function types:

If $A_2 <: A_1$ and $B_1 <: B_2$, then

$$A_1 => B_1 <: A_2 => B_2$$
So functions are *contravariant* in their argument type(s) and *covariant* in their result type.

This leads to the following revised definition of the `Function1` trait:

```scala
package scala
trait Function1[-T, +U] {
  def apply(x: T): U
}
```
We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```java
class Array[+T] {
    def update(x: T) ...
}
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`. 

---

**Variance Checks**

We have seen in the array example that the combination of covariance with certain operations is unsound.

In this case the problematic operation was the update operation on an array.

If we turn `Array` into a class, and `update` into a method, it would look like this:

```java
class Array[+T] {
    def update(x: T) ...
}
```

The problematic combination is

- the covariant type parameter `T`
- which appears in parameter position of the method `update`. 

---
The Scala compiler will check that there are no problematic combinations when compiling a class with variance annotations. Roughly,

- **covariant** type parameters can only appear in method results.
- **contravariant** type parameters can only appear in method parameters.
- **invariant** type parameters can appear anywhere.

The precise rules are a bit more involved, fortunately the Scala compiler performs them for us.
Variance-Checking the Function Trait

Let’s have a look again at Function1:

```scala
trait Function1[-T, +U] {  
  def apply(x: T): U
}
```

Here,

- T is contravariant and appears only as a method parameter type
- U is covariant and appears only as a method result type

So the method is checks out OK.
Variance and Lists

Let’s get back to the previous implementation of lists.

One shortcoming was that `Nil` had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make `List` covariant.
Variance and Lists

Let’s get back to the previous implementation of lists.

One shortcoming was that Nil had to be a class, whereas we would prefer it to be an object (after all, there is only one empty list).

Can we change that?

Yes, because we can make List covariant.

Here are the essential modifications:

```scala
trait List[+T] { ... }
object Empty extends List[Nothing] { ... }
```
Making Classes Covariant

Sometimes, we have to put in a bit of work to make a class covariant.

Consider adding a prepend method to List which prepends a given element, yielding a new list.

A first implementation of prepend could look like this:

```scala
trait List[+T] {
    def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

But that does not work!
Exercise

Why does the following code not type-check?

```scala
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

1. prepend turns List into a mutable class.
2. prepend fails variance checking.
3. prepend’s right-hand side contains a type error.
Exercise

Why does the following code not type-check?

```scala
trait List[+T] {
  def prepend(elem: T): List[T] = new Cons(elem, this)
}
```

Possible answers:

0 prepend turns List into a mutable class.
0 prepend fails variance checking.
0 prepend's right-hand side contains a type error.
Indeed, the compiler is right to throw out List with prepend, because it violates the Liskov Substitution Principle:

Here’s something one can do with a list \( xs \) of type \( \text{List}[\text{IntSet}] \):

\[
xs.\text{prepend}(\text{Empty})
\]

But the same operation on a list \( ys \) of type \( \text{List}[\text{NonEmpty}] \) would lead to a type error:

\[
ys.\text{prepend}(\text{Empty})
^\text{type mismatch}
\quad\text{required: NonEmpty}
\quad\text{found: Empty}
\]

So, \( \text{List}[\text{NonEmpty}] \) cannot be a subtype of \( \text{List}[\text{IntSet}] \).
Lower Bounds

But prepend is a natural method to have on immutable lists!

**Question**: How can we make it variance-correct?
But prepend is a natural method to have on immutable lists!

**Question**: How can we make it variance-correct?

We can use a *lower bound*:

```python
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

This passes variance checks, because:

- covariant type parameters may appear in lower bounds of method type parameters
- contravariant type parameters may appear in upper bounds of method
Exercise

Implement prepend as shown in trait List.

```scala
def prepend [U >: T] (elem: U): List[U] = new Cons(elem, this)
```

What is the result type of this function:

```scala
def f(xs: List[NonEmpty], x: Empty) = xs prepend x
```

Possible answers:

- 0 does not type check
- 0 List[NonEmpty]
- 0 List[Empty]
- 0 List[IntSet]
- 0 List[Any]
Exercise

Implement prepend as shown in trait List.

\[
\text{def prepend } [\text{U } >: \text{T}](\text{elem: U}): \text{List[U]} = \text{new Cons(elem, this)}
\]

What is the result type of this function:

\[
\text{def f(xs: List[NonEmpty], x: Empty) } = \text{xs prepend x}
\]

Possible answers:

0 does not type check
0 List[NonEmpty]
0 List[Empty]
0 List[IntSet]
0 List[Any]