香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 5)

COMP 3031, Fall 2025

Lionel Parreaux

Subtyping and Generics

Polymorphism

Two principal forms of polymorphism:

- subtyping
- generics

In this session we will look at their interactions.

Two main areas:

- bounds
- variance

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

Type Bounds

Consider the method `assertAllPos` which

- takes an `IntSet`
- returns the `IntSet` itself if all its elements are positive
- throws an exception otherwise

What would be the best type you can give to `assertAllPos`? Maybe:

```
def assertAllPos(s: IntSet): IntSet
```

In most situations this is fine, but can one be more precise?

Type Bounds

One might want to express that `assertAllPos` takes `Empty` sets to `Empty` sets and `NonEmpty` sets to `NonEmpty` sets.

A way to express this is:

```
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, “`<: IntSet`” is an *upper bound* of the type parameter `S`:

It means that `S` can be instantiated only to types that conform to `IntSet`.

Generally, the notation

- `S <: T` means: `S` *is a subtype of* `T`, and
- `S >: T` means: `S` *is a supertype of* `T`, or `T` *is a subtype of* `S`.

Lower Bounds

You can also use a lower bound for a type variable.

Example

```
[S >: NonEmpty]
```

introduces a type parameter `S` that can range only over *supertypes* of `NonEmpty`.

So `S` could be one of `NonEmpty`, `IntSet`, `AnyRef`, or `Any`.

We will see in the next session examples where lower bounds are useful.

Mixed Bounds

Finally, it is also possible to mix a lower bound with an upper bound.

For instance,

```
[S >: NonEmpty <: IntSet]
```

would restrict `S` any type on the interval between `NonEmpty` and `IntSet`.

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Covariance

There's another interaction between subtyping and type parameters we need to consider. Given:

```
NonEmpty <: IntSet
```

is

```
List[NonEmpty] <: List[IntSet]    ?
```

Intuitively, this makes sense: A list of non-empty sets is a special case of a list of arbitrary sets.

We call types for which this relationship holds *covariant* because their subtyping relationship varies with the type parameter.

Does covariance make sense for all types, not just for `List`?

Arrays

For perspective, let's look at arrays in Java (and C#).

Reminder:

- An array of `T` elements is written `T[]` in Java.
- In Scala we use parameterized type syntax `Array[T]` to refer to the same type.

Arrays in Java are covariant, so one would have:

```
NonEmpty[] <: IntSet[]
```

Array Typing Problem

But covariant array typing causes problems.

To see why, consider the Java code below.

```
NonEmpty[] a = new NonEmpty[]{
  new NonEmpty(1, new Empty(), new Empty())};
IntSet[] b = a;
b[0] = new Empty();
NonEmpty s = a[0];
```

It looks like we assigned in the last line an `Empty` set to a variable of type `NonEmpty`!

What went wrong?

The Liskov Substitution Principle

The following principle, stated by Barbara Liskov, tells us when a type can be a subtype of another.

> *If* `A <: B`*, then everything one can to do with a value of type* `B` *one should also be able to do with a value of type* `A`*.*

[The actual definition Liskov used is a bit more formal. It says:

> *Let* `q(x)` *be a property provable about objects* `x` *of type* `B`*. Then* `q(y)` *should be provable for objects* `y` *of type* `A` *where* `A <: B`*.]*

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
O	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
X	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Exercise

The problematic array example would be written as follows in Scala:

```
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

When you try out this example, what do you observe?

O	A type error in line 1
X	A type error in line 2
O	A type error in line 3
O	A type error in line 4
O	A program that compiles and throws an exception at run-time
O	A program that compiles and runs without exception

Variance in Scala

Variance

We saw that some types should be covariant whereas others should not.

Usually, types allowing mutation of their elements should not be covariant.

Immutable types can be covariant when 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]`	`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*

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

`class C[+A] { ... }`	`C` is *covariant*
`class C[-A] { ... }`	`C` is *contravariant*
`class C[A] { ... }`	`C` is *nonvariant*

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Int => Fruit` a subtype of `Int => Orange`, or the opposite?

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Int => Fruit` a subtype of `Int => Orange`, or the opposite?

Consider some function `buyFruit: Int => Fruit`;
can it be used anywhere function `buyOrange: Int => Orange` can be used?

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Int => Fruit` a subtype of `Int => Orange`, or the opposite?

Consider some function `buyFruit: Int => Fruit`;
can it be used anywhere function `buyOrange: Int => Orange` can be used?

- No: a counter example would be `pressOrange(buyOrange(42))`
  as `pressOrange(buyFruit(42))` would not make sense

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Int => Fruit` a subtype of `Int => Orange`, or the opposite?

Consider some function `buyFruit: Int => Fruit`;
can it be used anywhere function `buyOrange: Int => Orange` can be used?

- No: a counter example would be `pressOrange(buyOrange(42))` as `pressOrange(buyFruit(42))` would not make sense

However, `buyOrange` can be used anywhere `buyFruit` can be.

- So `Int => Orange <: Int => Fruit` ie, *functions are covariant* in their result type

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Fruit => Juice` a subtype of `Orange => Juice`, or the opposite?

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Fruit => Juice` a subtype of `Orange => Juice`, or the opposite?

Consider some function `pressFruit: Fruit => Juice`;
can it be used anywhere function `pressOrange: Orange => Juice` can be?

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Fruit => Juice` a subtype of `Orange => Juice`, or the opposite?

Consider some function `pressFruit: Fruit => Juice`;
can it be used anywhere function `pressOrange: Orange => Juice` can be?

- Yes: a function able to make juice out of any fruit can obviously be used to make juice out of oranges in particular.

Exercise

```scala
// Assume the following definitions
trait Fruit;  class Apple  extends Fruit
trait Juice;  class Orange extends Fruit
def pressOrange(o: Orange): Juice
```

- Is `Fruit => Juice` a subtype of `Orange => Juice`, or the opposite?

Consider some function `pressFruit: Fruit => Juice`;
can it be used anywhere function `pressOrange: Orange => Juice` can be?

- Yes: a function able to make juice out of any fruit can obviously be used to make juice out of oranges in particular.

However, `pressOrange` *cannot* be used anywhere `pressFruit` can be;
counter example: `pressFruit(new Apple)` and `pressOrange(new Apple)`

- So `Fruit => Juice <: Orange => Juice`
  ie, *functions are contravariant* in their input types

Exercise

Assume the following type hierarchy and two function types:

```
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type FtoO = Fruit => Orange
type AtoF = Apple => Fruit
```

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

```
O          FtoO <: AtoF
O          AtoF <: FtoO
O          A and B are unrelated.
```

Exercise (Solution)

Assume the following type hierarchy and two function types:

```
trait Fruit
class Apple extends Fruit
class Orange extends Fruit

type FtoO = Fruit => Orange
type AtoF = Apple => Fruit
```

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

```
X       FtoO <: AtoF
O       AtoF <: FtoO
O       A and B are unrelated.
```

Typing Rules for Functions

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

If `A2 <: A1` and `B1 <: B2`, then

```
A1 => B1  <:  A2 => B2
```

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:

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

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 array update operation.

Considering `Array` as a class and `update` as a method, it looks like this:

```
class Array[+T]:
  def update(idx: Int, x: T): Unit = ...
```

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 array update operation.

Considering `Array` as a class and `update` as a method, it looks like this:

```
class Array[+T]:
  def update(idx: Int, x: T): Unit = ...
```

The problematic combination is

- the purportedly-covariant type parameter `T`
- which appears in *input* (i.e., *parameter*) position of 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 array update operation.

Considering `Array` as a class and `update` as a method, it looks like this:

```
class Array[+T]:
  def update(idx: Int, x: T): Unit = ...
```

The problematic combination is

- the purportedly-covariant type parameter `T`
- which appears in *input* (i.e., *parameter*) position of method `update`

Scala would reject this definition.

Variance Checks (2)

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:

```
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).

```
sealed abstract class List[T] { ... }
class Empty[T] extends List[T] { ... }
```

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).

```
sealed abstract class List[T] { ... }
class Empty[T] extends List[T] { ... }
```

Can we change that? Yes, because we can make `List` covariant.

Here are the essential modifications:

```
sealed abstract class List[+T] { ... }
object Empty extends List[Nothing] { ... }
```

It works because `List[Nothing] <: List[T]` for any `T`

Idealized Lists

Here a definition of lists that implements all the cases we have seen so far:

```
sealed abstract class List[+T]:

  def isEmpty = this match
    case Nil => true
    case _ => false

  override def toString =
    def recur(prefix: String, xs: List[T]): String = xs match
      case x :: xs1 => s"$prefix$x${recur(", ", xs1)}"
      case Nil => ")"
    recur("List(", this)
```

Idealized Lists(2)

```scala
case class Cons[+T](head: T, tail: List[T]) extends List[T]
case object Nil extends List[Nothing]

extension [T](x: T) def :: (xs: List[T]): List[T] = Cons(x, xs)

object List:
  def apply() = Nil
  def apply[T](x: T) = x :: Nil
  def apply[T](x1: T, x2: T) = x1 :: x2 :: Nil
  ...
```

(We'll see later how to do with just a single `apply` method using a *vararg* parameter.)

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:

```
trait List[+T]:
  def prepend(elem: T): List[T] = elem :: this
```

But that does not work!

Exercise

Why does the following code not type-check?

```
trait List[+T]:
  def prepend(elem: T): List[T] = elem :: this
```

Possible answers:

O `prepend` turns `List` into a mutable class.

O `prepend` fails variance checking.

O `prepend`'s right-hand side contains a type error.

Exercise

Why does the following code not type-check?

```
trait List[+T]:
  def prepend(elem: T): List[T] = elem :: this
```

Possible answers:

O `prepend` turns `List` into a mutable class.

X `prepend` fails variance checking.

O `prepend`'s right-hand side contains a type error.

Prepend Violates LSP

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 `List[Fruit]`:

```
xs.prepend(Orange)
```

But the same operation on a list `ys` of type `List[Apple]` would lead to a type error:

```
ys.prepend(Orange)
           ^ type mismatch
           required: Apple
           found    : Orange
```

So, `List[Apple]` cannot be a subtype of `List[Fruit]`.

Lower Bounds

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

Q: How can we make it variance-correct?

Lower Bounds

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

Q: How can we make it variance-correct?

We can use a *lower bound*:

```
def prepend [U >: T] (elem: U) : List[U] = 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*.

Exercise

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = elem :: this
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)    ?
```

Possible answers:

O	does not type check
O	`List[Apple]`
O	`List[Orange]`
O	`List[Fruit]`
O	`List[Any]`

Exercise

Assume `prepend` in trait `List` is implemented like this:

```
def prepend [U >: T] (elem: U): List[U] = elem :: this
```

What is the result type of this function:

```
def f(xs: List[Apple], x: Orange) = xs.prepend(x)    ?
```

Possible answers:

O	does not type check
O	`List[Apple]`
O	`List[Orange]`
X	`List[Fruit]`
O	`List[Any]`

Extension Methods

The need for a lower bound was essentially to decouple the new parameter of the class and the parameter of the newly created object. Using an extension method such as in `::` above, sidesteps the problem and is often simpler:

```
extension [T](x: T)
  def :: (xs: List[T]): List[T] = x :: xs
```

A Closer Look At Lists

Lists Recap

Lists are the core data structure we will work with over the next weeks.

Type: `List[Fruit]`

Construction:

```
val fruits = List("Apple", "Orange", "Banana")
val nums = 1 :: 2 :: Nil
```

Decomposition:

```
fruits.head      // "Apple"
nums.tail        // 2 :: Nil
nums.isEmpty     // false

nums match
  case x :: y :: _ => x + y    // 3
```

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element; *exception* if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one; *exception* if `xs` is empty.
`xs.take(n)`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs.drop(n)`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs.apply(n)`). The element of `xs` at index `n`; *exception* if the index is invalid.

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element; *exception* if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one; *exception* if `xs` is empty.
`xs.take(n)`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs.drop(n)`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs.apply(n)`). The element of `xs` at index `n`; *exception* if the index is invalid.

It is generally better to avoid using *partial* methods
i.e., methods that can throw *exceptions*.

List Methods (2)

Creating new lists:

`xs ::: ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs.updated(n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`; *exception* if the index is invalid.

Finding elements:

`xs.indexOf(x)`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs.contains(x)`	same as `xs.indexOf(x) >= 0`

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case Nil => throw Error("last of empty list")
  case x :: Nil =>
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case Nil => throw Error(”last of empty list”)
  case x :: Nil => x
  case y :: ys =>
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case Nil => throw Error("last of empty list")
  case x :: Nil => x
  case y :: ys => last(ys)
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match
  case Nil => throw Error(”last of empty list”)
  case x :: Nil => x
  case y :: ys => last(ys)
```

So, `last` takes steps proportional to the length of the list `xs`.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case Nil => throw Error(”init of empty list”)
  case x :: Nil => ???
  case y :: ys => ???
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case Nil => throw Error(”init of empty list”)
  case x :: Nil =>
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case Nil => throw Error(”init of empty list”)
  case x :: Nil => Nil
  case y :: ys =>
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match
  case Nil => throw Error("init of empty list")
  case x :: Nil => Nil
  case y :: ys => y :: init(ys)
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`:

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil =>
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 =>
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: xs1 ::: ys
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: xs1 ::: ys
```

Note: as usual, '`x :: xs1 ::: ys`' parses as '`x :: (xs1 ::: ys)`'

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: xs1 ::: ys
```

Note: as usual, '`x :: xs1 ::: ys`' parses as '`x :: (xs1 ::: ys)`'

What is the complexity of `concat`?

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing an extension method for `:::`

```
extension [T](xs: List[T])
  def ::: (ys: List[T]): List[T] = xs match
    case Nil => ys
    case x :: xs1 => x :: xs1 ::: ys
```

Note: as usual, '`x :: xs1 ::: ys`' parses as '`x :: (xs1 ::: ys)`'

What is the complexity of `concat`?

Answer: `O(xs.length)`

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil =>
    case y :: ys =>
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys =>
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ::: y :: Nil
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ::: y :: Nil
```

What is the complexity of `reverse`?

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing an extension method:

```
extension [T](xs: List[T])
  def reverse: List[T] = xs match
    case Nil => Nil
    case y :: ys => ys.reverse ::: y :: Nil
```

What is the complexity of `reverse`?

Answer: `O(xs.length * xs.length)`

Can we do better? (to be solved later).

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```
> *List(a, c, d)*

Exercise

Remove the n'th element of a list xs. If n is out of bounds, return xs itself.

```
def removeAt[T](n: Int, xs: List[T]) = xs match
  case Nil => Nil
  case y :: ys =>
    if n == 0 then ys
    else y :: removeAt(n - 1, ys)
```

Exercise (Harder, Optional)

"Deeply" flatten a list structure:

```
def deepFlatten(xs: Any): List[Any] = ???
```

```
 deepFlatten(List(List(1, 1), 2, List(3, List(5, 8))))
             >   res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```

Exercise (Harder, Optional)

"Deeply" flatten a list structure:

```
def deepFlatten(xs: Any): List[Any] = xs match
  case Nil => Nil
  case y :: ys => deepFlatten(y) ::: deepFlatten(ys)
  case nonList => nonList :: Nil
```