# Polymorphism

Principles of Functional Programming

# Cons-Lists

A fundamental data structure in many functional languages is the immutable linked list.

It is constructed from two building blocks:

| | |
|---|---|
| `Nil` | the empty list |
| `Cons` | a cell containing an element and the remainder of the list. |

# Examples for Cons-Lists

```
List(1, 2, 3)
```

```
List(List(true, false), List(3))
```

# Cons-Lists in Scala

Here's an outline of a class hierarchy that represents lists of integers in this fashion:

```
package week3

trait IntList ...
class Cons(val head: Int, val tail: IntList) extends IntList ...
class Nil() extends IntList ...
```

A list is either

- an empty list `Nil()`, or
- a list `Cons(x, xs)` consisting of a `head` element `x` and a `tail` list `xs`.

# Value Parameters

Note the abbreviation `(val head: Int, val tail: IntList)` in the definition of `Cons`.

This defines at the same time parameters and fields of a class.

It is equivalent to:

```
class Cons(_head: Int, _tail: IntList) extends IntList:
  val head = _head
  val tail = _tail
```

where `_head` and `_tail` are otherwise unused names.

# Type Parameters

It seems too narrow to define only lists with `Int` elements.

We'd need another class hierarchy for `Double` lists, and so on, one for each possible element type.

We can generalize the definition using a type parameter:

```
package week3

trait List[T]
class Cons[T](val head: T, val tail: List[T]) extends List[T]
class Nil[T] extends List[T]
```

Type parameters are written in square brackets, e.g. `[T]`.

# Complete Definition of List

```
trait List[T]:
  def isEmpty: Boolean
  def head: T
  def tail: List[T]

class Cons[T](val head: T, val tail: List[T]) extends List[T]:
  def isEmpty = false

class Nil[T] extends List[T]:
  def isEmpty = true
  def head = throw new NoSuchElementException("Nil.head")
  def tail = throw new NoSuchElementException("Nil.tail")
```

# Generic Functions

Like classes, functions can have type parameters.

For instance, here is a function that creates a list consisting of a single element.

```
def singleton[T](elem: T) = Cons[T](elem, Nil[T])
```

We can then write:

```
singleton[Int](1)
singleton[Boolean](true)
```

# Type Inference

In fact, the Scala compiler can usually deduce the correct type parameters from the value arguments of a function call.

So, in most cases, type parameters can be left out. You could also write:

```
singleton(1)
singleton(true)
```

# Types and Evaluation

Type parameters do not affect evaluation in Scala.

We can assume that all type parameters and type arguments are removed before evaluating the program.

This is also called *type erasure*.

Languages that use type erasure include Java, Scala, Haskell, ML, OCaml.

Some other languages keep the type parameters around at run time, these include C++, C#, F#.

# Polymorphism

Polymorphism means that a function type comes “in many forms”.

In programming it means that

- the function can be applied to arguments of many types, or
- the type can have instances of many types.

# Polymorphism

Polymorphism means that a function type comes “in many forms”.

In programming it means that

- the function can be applied to arguments of many types, or
- the type can have instances of many types.

We have seen two principal forms of polymorphism:

- subtyping: instances of a subclass can be passed to a base class
- generics: instances of a function or class are created by type parameterization.

# Exercise

Write a function `nth` that takes a list and an integer `n` and selects the `n`'th element of the list.

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

Elements are numbered from 0.

If index is outside the range from `0` up the the length of the list minus one, a `IndexOutOfBoundsException` should be thrown.