Polymorphism

Cons-List

- 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
- Lists can also contain lists and can also contain lists of different types
- Here’s an outline of a class hierarchy that represents lists of integers in this fashion:

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

- A list is either:
    - an empty list ***Nil***
    - a list ***Cons(x, xs)*** consisting of a head element x and a tail list xs

Value Parameters

- The abbreviation ***(val head: Int, val tail: IntList)*** in the definition of ***Cons*** 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:

```
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

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 write: ***singleton[Int](1), singleton[Boolean](true)***

Type Inference

- 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

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*** (types get erased during the compilation process)

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 parametrization

Exercise: *Write a function that takes a list and an integer n and selects the n^{th} element of the list. Elements are numbered from 0. If index is outside the range from 0 up to the length of the list minus one, a* ***IndexOutOfBoundsException*** *should be thrown.*

```
def nth[T](xs: List[T], n: Int): T =
  if (xs.isEmpty)
    throw IndexOutOfBoundsException()
  else
     if (n == 0)
       xs.head
     else
       nth(xs.tail, n – 1)
```