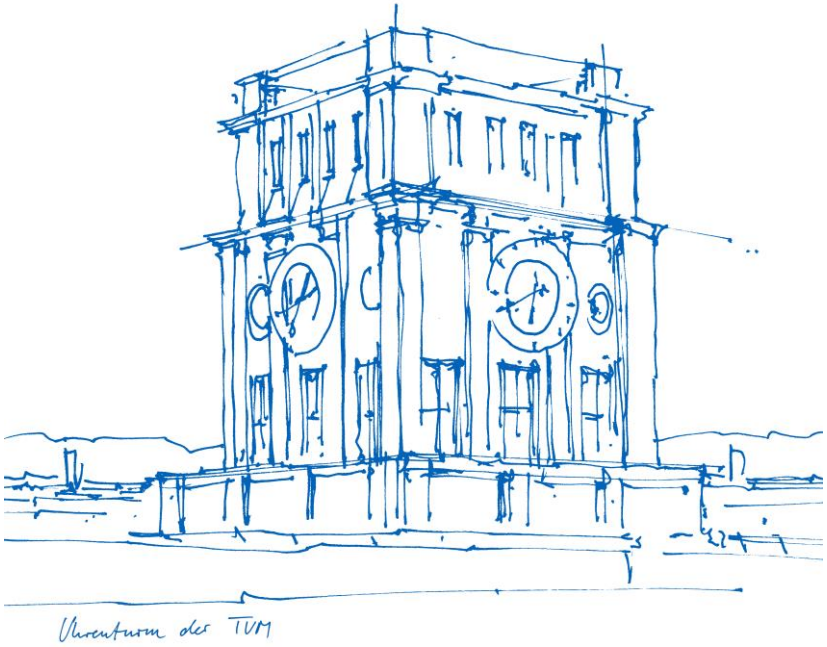
Future Trends in Computing
X10 Basics – Sequential Programming

Alexander Pöppl, Emily Ao Mo-Hellenbrand

Technical University of Munich

Department of Informatics

Chair of Scientific Computing in Computer Science



Conditionals and Loops

If statement like in other C-like languages:

```
if (cond) {
        //code
} else {
        //more code
}
```

Same for while:

```
while (cond) {
        //code
}
```

And the standard for loop:

```
for (var x:Long = 0; x < 10; x++) {
        //Code
}
```

Chair of Scientific Computing in Computer Science
Department of Informatics
Technical University of Munich

Of vars and vals

X10 has notion of mutable & immutable values
Default should be immutable **values**

- Type inferred automatically when immediately initialized.

```
val x = 5.0;                          // Double
val y = 6.8f;                         // Float
val z = 10n;                          // Int
val a = 42;                           // Long
val b = "I am a String";              // String
val d = Place(0);                     // Place
val c = new Rail[Float](3000);        // Rail[Float]{size==3000}
```

- May be initialized later, then type annotation neccesary. Must be initialized before use!

```
val late : Long;
late = 4;
```

- May not be reassigned.

```
late = 5;
```

Of vars and vals

X10 has notion of mutable & immutable values
Mutable variables are supported too:

No type inference for variables!

```
var x : Long = 0;
var y = 42;
var z : Long;
z = 4;
z = 5;
```

Type System

X10 is strongly and statically typed

Everything is an object, no primitive values like in Java or C/C++
Objects are instances of **class**es or **struct**s.

```
42.toBinaryString(); //101010
```

It is possible to parameterize types

```
val x : Rail[String];
val y : Pair[Int, Pair[Float, String]];
```

And to constrain the values they may assume:

```
val z : Float{self==0.9f};
val r : Rail[Boolean]{size==32};
```

Functions types are also possible:

```
val f : (Float, Int) => Float;
```

Structs vs. Classes

```
public class HelloWorld {
        // Code
}
```

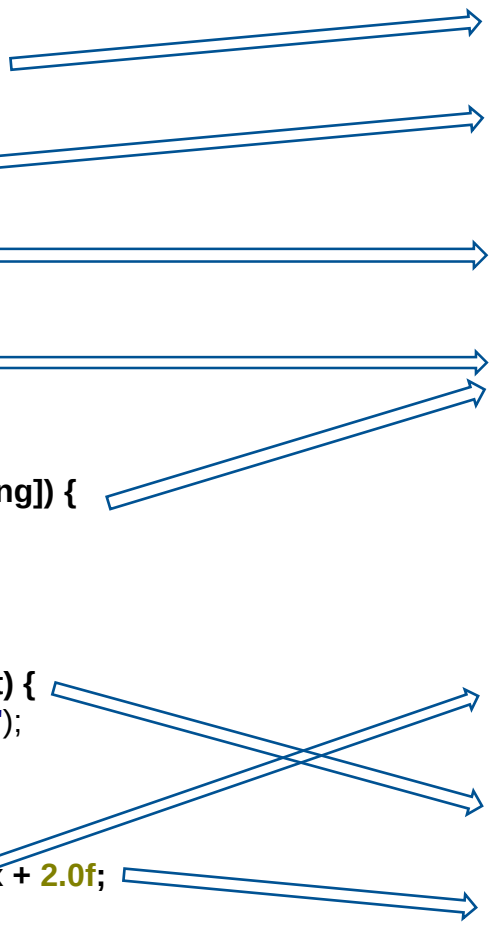
Reference type
Mutable and immutable members
Heap-Allocated
Passed-by-reference

```
public struct HelloWorld {
        // Code
}
```

Value type
Only immutable member types
Stack-allocated (Native Backend)
Passed-by-Value

Anatomy of an X10 Class/Struct

```
public class HelloWorld {

  static val SOME_CONSTANT = 2;

  val x : Float;
  val y : Rail[String];

  var z : Int;

  public def this() {
    this(0.0f, ["a", "b"]);
  }

  public def this (x:Float, y:Rail[String]) {
    this.x = x;
    this.y = y;
  }

  public def instanceMethod(a:Float) {
    Console.OUT.println("a" + " * " + "x");
    return a*x;
  }

  public operator this -> (f:Float) = x + 2.0f;
  public def getX() = x;
}
```



- Static member (constant)
- Instance Members (immutable)
- Instance Members (mutable)
- Constructors
- Call to other constructor
- Instance methods
- Instance method (short form)
- Instance method (long form)
- Overloaded Operator

Higher-Order Functions

Sometimes taking a function as a parameter may be useful:

```
public def filter[T](predicate:(T)=>Boolean, array:Rail[T]) : Rail[T] {
  //Implementation
}
```

X10 supports anonymous functions as first-class citizens.

```
val f = (t:Float) => t >= 1.0f;
```

They may be created and passed around freely

```
val b = f(2.0f);
val filtered = filter(f, [1.0f, 3.5f, 5.5f]);
```

Arrays

X10 has three fundamental Array styles:

Basic, zero-based, one-dimensional Array: **Rail[T]**

```
val r1 = new Rail[Double](10);
val r2 = new Rail[Double](10, (i:Long)=>i*Math.PI);
val r3 = [1.3, 4.5, 6.3];
val a = r2(4);
r1(2) = a;
```

More sophisticated, zero-based, multidimensional Arrays: package **x10.array**

```
val b = new Array_2[Double](N, N, (i:Long,j:Long)=>(i-j) as Double);
r3 = b(0,3);
```

Point-based, freely definable arrays of arbitrary shape and size: package **x10.regionarray**

- Later

Arrays (cont'd.)

X10 has shorthand code for array iteration:

```
var b = new Array_2[Double](N, N, (i:Long, j:Long) => (i-j) as Double);

for (var i:Long=0; i<b.numElems_1; i++) {
  for (var j:Long=0; j<b.numElems_2; j++) {
    sum += b(i,j);
  }
}

for (i in 0..(N-1)) {
  for (j in 0..(N-1)) {
    sum += b(i,j);
  }
}

for ([i,j] in b.indices()) {
  sum += b(i,j);
}

for (v in b) {
  sum += v;
}
```

Arrays (cont'd.)

Arrays in x10.regionarray package depend on Points and Regions

Points are n-dimensional, integral tuples

```
val origin_1 : Point{rank==1} = Point.make(0);
val origin_2 : Point{rank==2} = Point.make(0,0);
val origin_5 : Point = Point.make(new Rail[Long](5));
val p : Point = [1,2,3];
val q : Point{rank==5} = [1,2,3,4,5];
val r : Point(3) = [11,22,33];
```

Regions are a set of points of the same rank (dimension). They serve as a "blueprint" of instances of the regionarray package's Array class.

```
val Null = Region.makeUnit(); //Empty 0-dimensional region
val R1 = Region.make(1, 100); // Region 1..100
val R2 = Region.make(1..100);  // Region 1..100
val R3 = Region.make(0..99, -1..MAX_HEIGHT);
val R4 = Region.makeUpperTriangular(10);
val R5 = R4 && R3; // intersection of two regions
```

Arrays (cont'd.)

Arrays in x10.regionarray package depend on Points and Regions

May be iterated over similarly to other Array class. Indexing using Point objects possible.

```
val A4 = new Array[Long](Region.make(1..2, 1..3, 1..4, 1..5), 0);
A4(2,3,4,5) = A4(1,1,1,1)+1;

for (p in A4) {
  A4(p) = p(0) + p(1) + p(2) + p(3);
}

for ([i,j,k,l] in A4) {
  A4(i,j,k,l) += i*j*k*l;
}
```

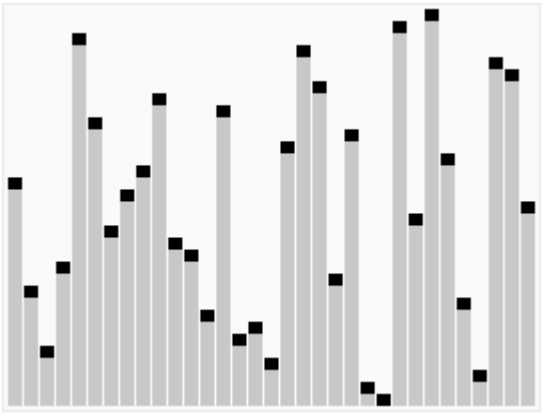
File I/O

IO in package x10.io.*

```
// FileReader

val reader = new FileReader(f);
for (line in reader.lines()) {
  //.. code
}

val writer = new FileWriter(f, append);
writer.write(…)
```

Exercise: Quicksort

- Comparison sort: can sort items of any type for which a "**<**" relation is defined



- Divide and Conquer:
  - Divide a large array into 2 sub-arrays
  - Recursively sort sub-arrays

- **Algorithm:**
  1. Pick an element as **Pivot**
  2. Partitioning: re-order the array such that
     all elements < **Pivot** come before the **Pivot**, and all elements > **Pivot** come after the **Pivot**.
  3. Recursively apply the partitioning step to the "smaller" and "greater" sub-arrays

- Different variations of Quicksort available
  e.g. Lomuto scheme: chooses last element as pivot

Exercise: Quicksort

- **Lomuto scheme**

```
algorithm quicksort(A, lo, hi) :
        if lo < hi then
                p := partition(A, lo, hi)
                quicksort(A, lo, p – 1)
                quicksort(A, p + 1, hi)
```

```
algorithm partition(A, lo, hi):
      pivot := A[hi]    // choose last element as pivot
      i := lo           // place for swapping
      for j := lo to hi – 1 do
          if A[j] ≤ pivot then
              swap A[i] with A[j]
              i := i + 1
      swap A[i] with A[hi]
      return i
```

Exercise: Quicksort

Implement a class Quicksorter that implements the Quicksort algorithm to sort instances of the struct MyComplex.

MyComplex is a struct that models a Complex number in Polar coordinates. It has

- A constructor taking the angle and the length
- A constructur taking the real and imaginary component
- Comparison operators $<, \leq, \geq, >$. This is done by implementing the interface Ordered<MyComplex>.
- A toString() method for pretty printing

Quicksorter is a class that offers a method that takes an array and sorts it in-place using the QuickSort algorithm. One may specify a custom comparison function, passed as a parameter.