Scala Methods & Functions 1

- use "def" for a method (both functions and procedures)
  - For functions, have “=“ separating fn header from body
  - e.g., def exampleFunction(int n) = { }
  - e.g. def performAction(int n) { }

- To turn a method fn created w/“def” into standard function for passing or returning as an argument, follow by _
  - This makes it clear that are not simply calling, but seeking to take its value as a function

Referring to Methods as Functions

```
scala> class Foo { var m = 2;    def incr() = { m += 1;   m } }
defined class Foo
scala> val foo = new Foo()
foo: Foo = Foo@419bc62d
scala> foo.incr
res4: Int = 3
scala> foo.incr _
res5: () => Int = <function0>

scala> floor _
res14: Double => Double = <function1>
```

This is a function that, when called, will increment foo

Refers to the “floor” function (rather than to the method itself)

Method Formal Parameters

- Can include **default values**
  - These values are used if no argument is given for that parameter when the function is called
  - e.g. def join(sep = “,”)
  - This can really cut down the amount of arguments that have to be passed. This offers greater:
    - Convenience
    - Clarity of the key argument values
  - This can also much reduce need for auxiliary constructors
- Can be **specified by name** when calling
  - Given specification by name, can give in any order
  - Can mix specification by just listing and then by name
- Can be specified as subject to *implicit* specification

Anonymous Functions 1

- Here, can write anonymous functions in syntax (a: Int) => (a*a), a => a*a
- Thus we can write

```
(0 to 10).map(i => i * 3)
```

```
(1 to 30).reduceRight((n, a) => n + a)
```

Succinct Notation for Anonymous Functions

- To avoid writing out full closure when just single argument and only appears once in body, can use "_" to denote parameter in the body alone

  e.g., (0 to 10).map(_* 3)

- If "_" appears in the body twice, it is assumed to represent two different values, e.g.,

  (1 to 30).reduceRight(_ + _)

Statically Defined **Methods** in Scala

```
def square(x: Int) = x*x

def increment(x: Int) = x+1
```

```
scala> square(3)
res6: Int = 9

scala> increment(2)
res5: Int = 3
```

To treat as a **function** (and obtain a corresponding function object referring to the method), follow by _

```
scala> square _
res7: Int => Int = <function1>
```

Anonymous Functions

- Some of the most powerful uses of closures occur with unnamed functions
  - This functions are created to accomplish some particular ad hoc or dynamic task
  - If the task is commonly needed, we can create a name function to abstract that functionality
- Anonymous functions play a very important role in functional programming for higher order fns
  - Passed to functions
  - Returned from functions
- What do these anonymous do?
  - (n:Int) => n+1
  - (n:Int) => n*2
  - scala> (c:Int) => ((n:Int) => n+c) // requires closure
  - res10: Int => (Int => Int) = <function1>

Creating Closures

- A closure binds a function to some scope (environment)
  - Environment: A mapping of names to locations
    - In a purely functional languages, this maps names to values
  - ***When we create a function value, a closure is created with it that captures the values for vars on which function relies***
  - e.g.
    - val IncrementByFunctionGenerator = (c:Int) => ((n:Int) => n+c) // a closure is formed when this (function) value is returned. This closure captures the "binding" between "c" and whatever value was passed as the argument to "foo"

Functions in Python

- **Parameterizing: Add parameterized (e.g. user-specified) constant to the elements**

```
val FnIncrementBy = (c:Int) => ((n:Int) => n+c)
val FnSimpleIncrement = FnIncrementBy(1)
val FnSimpleIncrement(5)
res1: Int = 6
val FnAdd10=FnIncrementBy(10)
val FnAdd10(5)
res2: Int = 15
val testArray = (-1 to 10)
scala> testArray.map(FnAdd10)
res3: scala.collection.immutable.IndexedSeq[Int] = Vector(9, 10, 11, 12, 13, 14,
  15, 16, 17, 18, 19, 20)
scala> testArray.map(FnSimpleIncrement)
res4: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7,
  8, 9, 10, 11)
```

Delay

- Wrapping in anonymous function w/ no args offers simple mechanism for delaying evaluation of code
- Allows capturing of some benefits of lazy evaluation

```
scala> val fn = (() => print("foo"))
fn: () => Unit = <function0>

scala> fn()
foo
```

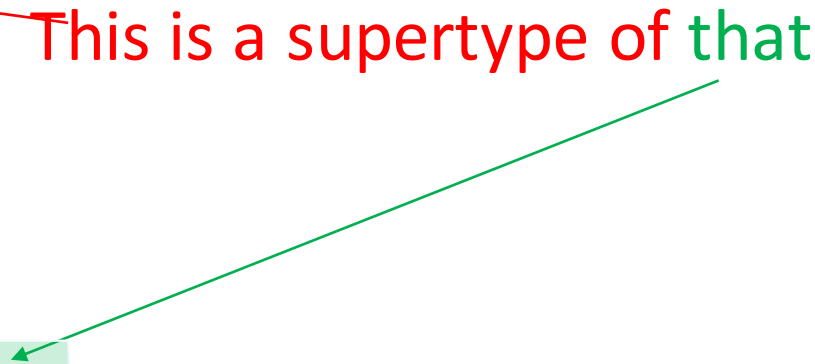
- Cf setting up event handlers
  - arrEventHandlers(iOnMouseSingleClick) = () => PerformCalc1()
  - arrEventHandlers(iOnMouseDoubleClick) = () => PerformCalc2()
- Cf Streams,BigNums
- In Scala, it can often be more natural to capture these with lazy values or call by name

Scala Functions: Contravariance & Covariance

- In contrast to traditional Java or C++, Scala supports more general type rules for functions motivated by the Liskov Substitution Principle
- By LSP, Implementation of a function/method *f* in subtype B of a supertype A must have
  - Contravariant in Parameters: a given parameter of *f* as defined in B must have a type that is equivalent or a supertypes of the corresponding parameter of *f* in supertype A
    - "Contra" in "Contravariant": As to go successive subtypes, the parameters can go to successive supertypes! (As go to successive specialization, parameters can become more general)
  - Covariant in Return values/exceptions: The return value and exceptions thrown by *f* in B can be the same or subtypes of those of *f* in A
- Scala supports the latter of these (specifically, covariant return values)

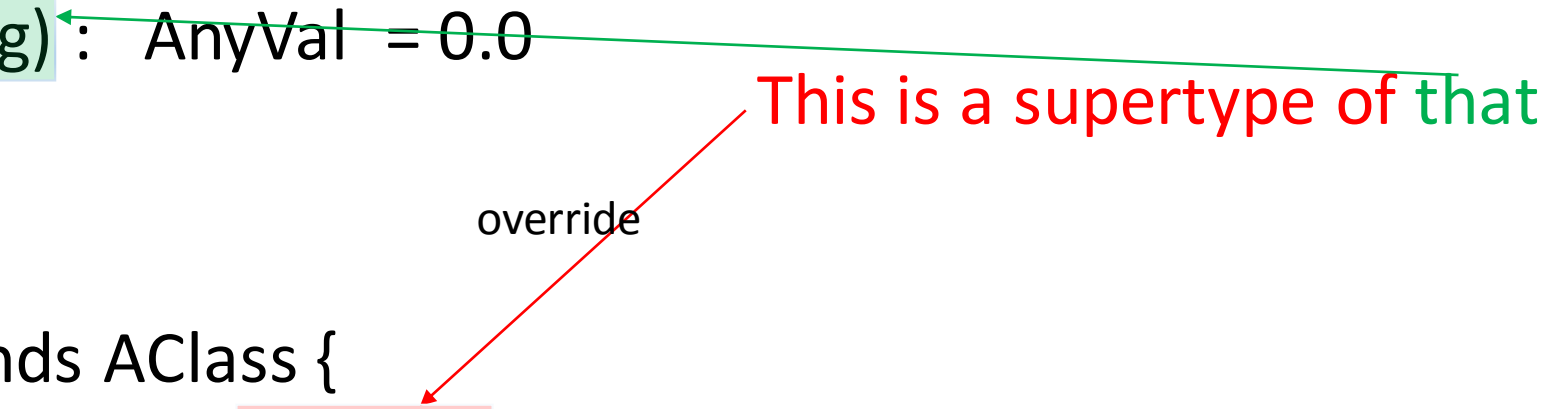
Acceptable to Scala Compiler

```
class AClass {
    def foo(v: AnyRef) : AnyVal = 0.0
}

class BClass extends AClass {
        override def foo(v: AnyRef) : Double =  0.0
}
```



Should be Acceptable, But Refused by Scala Compiler

```
class AClass {
    def foo(v: String) :   AnyVal  = 0.0
}

class BClass extends AClass {
        override def foo(v: AnyRef) : Double =  0.0
}
```



This satisfies the Liskov Substitution Principle, but is still refused by the Scala compiler

Apparent Danger to the Liskov Substitution Principle with Scala Self Types

```
class AClass {
   def bar(v: this.type) : AnyVal  = 0.0  }
class BClass extends AClass {
   override def bar(v: this.type) : AnyVal  = 0.0  // This should not be allowed! }
```

- Ok
  - val a: AClass = new AClass()
  - val b: BClass = new BClass()
  - val apparentAClass : Aclass
  - a.bar(a)
  - b.bar(a)
  - val apparentAClass : AClass = b
- Should work but doesn't
  - apparentAClass.bar(a)
  - apparentAClass.bar(apparentAClass)

This is a "self type", which refers to the type of the enclosing class. By the L SP, Scala should not allow a subtype Method using this as a parameter to override a corresponding supertype method, but does.
This leaves the door open to a gratuitous violation of the LSP!

Scala Functions 2

- Note that can have methods that are themselves polymorphic
- Can be useful to use “this.type” to denote surrounding type
  - For example, making the return value of an overridden function (method) “this.type” allows subtypes to return corresponding subtypes (e.g., subtype A return A) -- covariance
- Can have nested functions: Functions whose scope is limited to and used within another function
- Currying: can explicitly define multiple lists for a function (in successive parentheses), so that provide arguments in staged fashion
  - MyAdd(x:Double)(y:Double)

Scala Functions 3

- Methods that only take a single argument (but which are also applied to some "target" as the method) can be used as infix operators
  - def plus(a: Int): Int = ….
  - e.g., a plus b
- Can use operators that don't take an argument can be used as a postfix operator
  - def complement(): Set = ….
  - e.g., ASet complement
- Above: *need care & s.t. () to ensure scope of each parameter (e.g., a fn) is clear*
- When call unary function, can omit
- can overload operator syntax, as in C++
  - (e.g., use syntax like "^"), but I believe that can't change precedence easily

Built-in Support for Tuples

- Elements can be of different types
- Constructed implicitly with parentheses
  - e.g., (“Pi”, 3.14159)
- With pattern matching, can put on left hand side of an assignment
  - E.g. val (a,b,c) = fooReturningMultipleValues()
- The underlying type parameterized classes are Tuple2, Tuple3, Tuple4, … Tuple22,etc.

Collections 1

- Scala provides rich set of collections, in 3 categories: seq, set, map
- Generally type parameterized (e.g., Array[T])
- Can create w/name of collection (e.g., Array(1,2,3), Map(“a”->1,”b”->2))
- Include all of
  - Lazy vs. not
  - Immutable vs. mutable
- Equality is based on
  - If in different category, then not equal
  - equality of elements (collections of different types/structures are unequal)
- Can use “par” method to render parallel version of collection
- Can use foreach syntax e.g.,
  - coll foreach { (a) => print(a); }

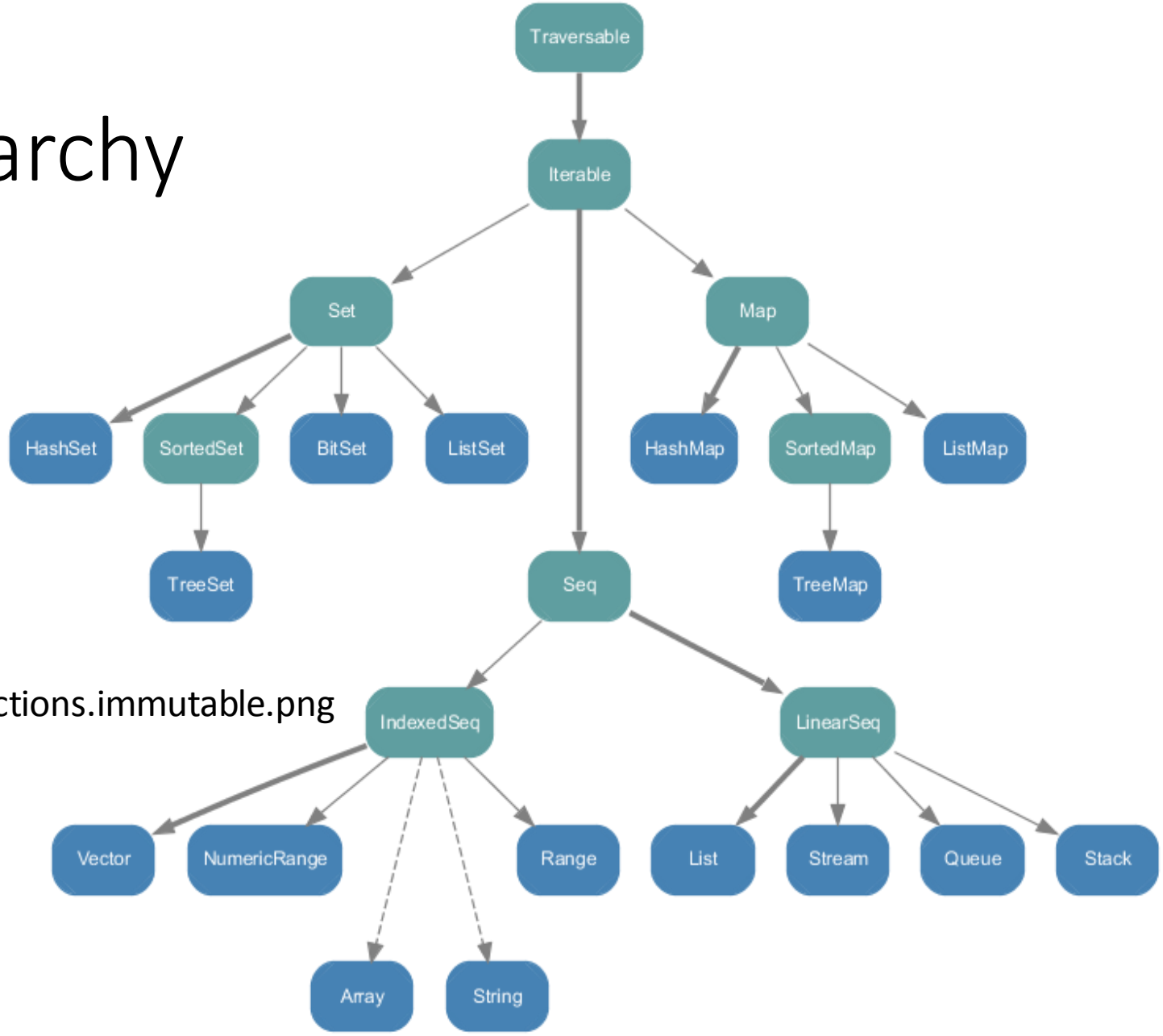
Collections 2: Example Collections

- **Stream[T]** lazy, use #:: to build from head/tail. Operations are **incremental**
- **Vector[T]**: O(1) time for operations on head, others take time linear to position/depth in sequence
- **Range**: Created w/"to" and "by" (e.g., 0 to 100 by 2) and until(m,n) (upper:n-1)
- **Array[T]** Familiar array
  - single dimensional: can allocate an array of Ints of size n using new Array[Int](n)
  - can access with array1D(i), add elements (single or tuple) to with +=, collection with ++=
  - multidimensional: Array.ofDim[Int](countRows, countColumns)
    - have then Array[Array[Int]]
    - access with array2(i)(j)
- **List[T]**: Can use :: operator to build from head/tail
- **Map[K, V]**: to create a map from a key to a value (e.g., String to a ComputeLivenessRule)
  - val myMap = Map("square" -> ((a: Int) => (a*a)),
    "cube" -> ((a: Int) => (a*a*a)))

Scala Maps

- Represent key-value mappings (Collections of key-value pairs)
- Can create with normal collection syntax: Map(“a”->1,”b”->2)
  - Note no “new” operator!
- For mutable map, can add new item (key k and value v) in either of the sort of way
  - map(k)=v (e.g., map(“Scala” = “Great!”))
  - map += (k -> v) (e.g., map += (“Scala” -> “Great!”))
- Can deconstruct via pattern matching as a (k,v) pair)
  - Iterate over map with (for (k,v) <- map) *body*

Collections Hierarchy



http://docs.scala-lang.org/resources/images/collections.immutable.png

Classic Functional Programming Constructs

- **Container[E].map: (E => F) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].flatMap: (E => Container[F]) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].reduce**: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter**: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Another: **Container[E].groupBy**: (E => K) => Map[K, **Container[E]]**
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

High-Level Description

- **Reduce** represents “accumulation” – independent of what exactly one is accumulating
- **Filter** represents conditional selection – independent of what exactly one is selecting (the exact rules for selection)
- **Map** represents a mapping (transformation) of each element of a collection

Classic Functional Programming Constructs

- **Container[E].map: (E => F) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].flatMap: (E => Container[F]) => Container[F]**
  - **Applies mapping function to each element of collection**
- **Container[E].reduce**: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter**: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Another: **Container[E].groupBy**: (E => K) => Map[K, **Container[E]]**
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Initial Binding

```
scala> val testArray = (-1 to 10)
testArray: scala.collection.immutable.Range.Inclusive = Range(-1, 0, 1,
2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Map

- **What does this do?**
  - scala> testArray.map(n => n * n)

- **And this?**
  - scala> testArray.map(n => n + 1)

  What is an alternative expression of the same function?

- **What function does the following transformation undertake?**
  scala> testArray.map(n => n*(if (n > 1) 1 else -1))

Map: (E→E)×Collection[E]→Collection[E]

- **Square the elements of the collection**
  - scala> testArray.map(n => n * n)
  - res8: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
- **Add fixed value (one) to each elements**
  - scala> testArray.map(n => n + 1)
  - res9: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

  An alternative expression of the same function:
  - scala> testArray.map(_ + 1)
  - res10: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
- **Value-specific mapping**

  scala> testArray.map(n => n*(if (n > 1) 1 else -1))

  res13: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 0, -1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Passing closures resulting from higher-order functions, and named closures

```
val FnIncrementBy = (c:Int) => ((n:Int) => n+c)

scala> testArray.map(FnIncrementBy(3))
res17: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
  12, 13)

val FnAdd10=FnIncrementBy(10)

scala> testArray.map(FnAdd10)
res18: scala.collection.immutable.IndexedSeq[Int] = Vector(9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20)
```

Classic Functional Programming Constructs

- Container[E].map: (E => F) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].flatMap: (E => Container[F]) => Container[F]
  - Applies mapping function to each element of collection
- **Container[E].reduce**: (F × E => F) => F
  - **Applies a function to each element of a collection and to the previous result of the reduce function**
- Container[E].filter: (E => Boolean) => Collection[E]
  - Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)
- Another: Container[E].groupBy: (E => K) => Map[K, Container[E]]
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Reduce

- **Totalling up the elements of the array**

```
>>> reduce(function(a,b) { return(a+b); }, rg)
  10
```

- Concatening array elements

```
>>>reduce(function(a,b) { return(a+""+b); }, rg)
```

- Taking max of elements

```
>>> reduce(function(x,y) { return (x>y) ? x : y; }, rg);
```

Variants

- Reduce
  - reduceLeft
  - reduceRight
- Fold
  - foldLeft
  - foldRight
  - “Curried” to take initial value

Classic Functional Programming Constructs

- Container[E].map: (E => F) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].flatMap: (E => Container[F]) => Container[F]
  - Applies mapping function to each element of collection
- Container[E].reduce: (F × E => F) => F
  - Applies a function to each element of a collection and to the previous result of the reduce function
- **Container[E].filter: (E => Boolean) => Collection[E]**
  - **Returns a collection consisting of elements of a given collection matching a given predicate (spec. via a function)**
- Another: Container[E].groupBy: (E => K) => Map[K, Container[E]]
  - Returns a map (e.g., dict) whose keys are each value of k produced by applying function to elements of the source container, and whose value for a given key k consists of a container of the values of the original container mapping to that key k

Filter

- scala> testArray.filter(n => n > 0)
- res19: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  Alternatively,
- scala> testArray.filter(_ > 0)
- res20: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- scala> testArray.filter(_ % 2 == 0)
- res21: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 2, 4, 6, 8, 10)

- Numbers between 0 and 5 (inclusive)
  - scala> testArray.filter(n => (n >= 0 && n <= 5))
  - res22: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 2, 3, 4, 5)