Concurrent Programming in Scala

- Advantages of Functional Programming
- Collections
  - Immutable
  - Parallel
- Composable Futures
- Reactive Programming
  - Observables

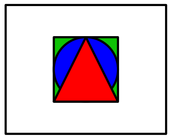
Models of Computation

- **Imperative: Step by step instructions**
  - Changing memory cells

⇨ do_f;

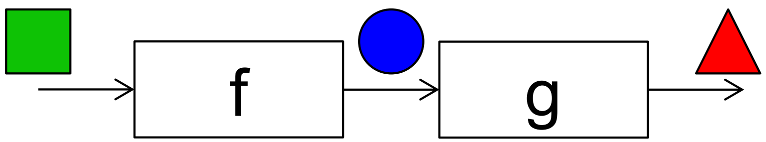
⇨ do_g;

var



```scala
class ImpCalc {
  private var a: Int = 0
  private var b: Int = 0

  def setA(a: Int) = this.a = a
  def setB(b: Int) = this.b = b

  def sum: Int = a + b
}
```

- **Functional: Applying functions to arguments**
  - Transforming data through pipelines of pure functions




```scala
object FunCalc {
  def sum(a: Int, b: Int): Int =
    a + b
}
```

Why Pure Functions are Great

- **Given an unknown pure function named xxx**

```
def xxx(i: Int): Int
```

- **What is the result / value of the following expression?**

```
xxx(42) – xxx(42)
```

Always 0! Because pure functions always return the same result when given the same arguments! ✓

Why Mutable Objects are Dangerous

- **In comparison take the following unknown Java method**

```
class X { ...
  public int xxx(int i) { ... }
}
```

- **What can be said about the result of the following expression?**

```
X x = ...;
x.xxx(42) - x.xxx(42)
```

Nothing! Because methods can behave differently on every call!

```
class X {
  private int cnt = 3;
  public int xxx(int i) {
    if(--cnt == 0) {
      killBambi();
      return i * cnt;
    }
    return i * 3;
  }
}
```

Concurrent Programming in Scala

- **Advantages of Functional Programming**
- **Collections**
  - Immutable
  - Parallel
- **Composable Futures**
- **Reactive Programming**
  - Observables

Fachhochschule Nordwestschweiz

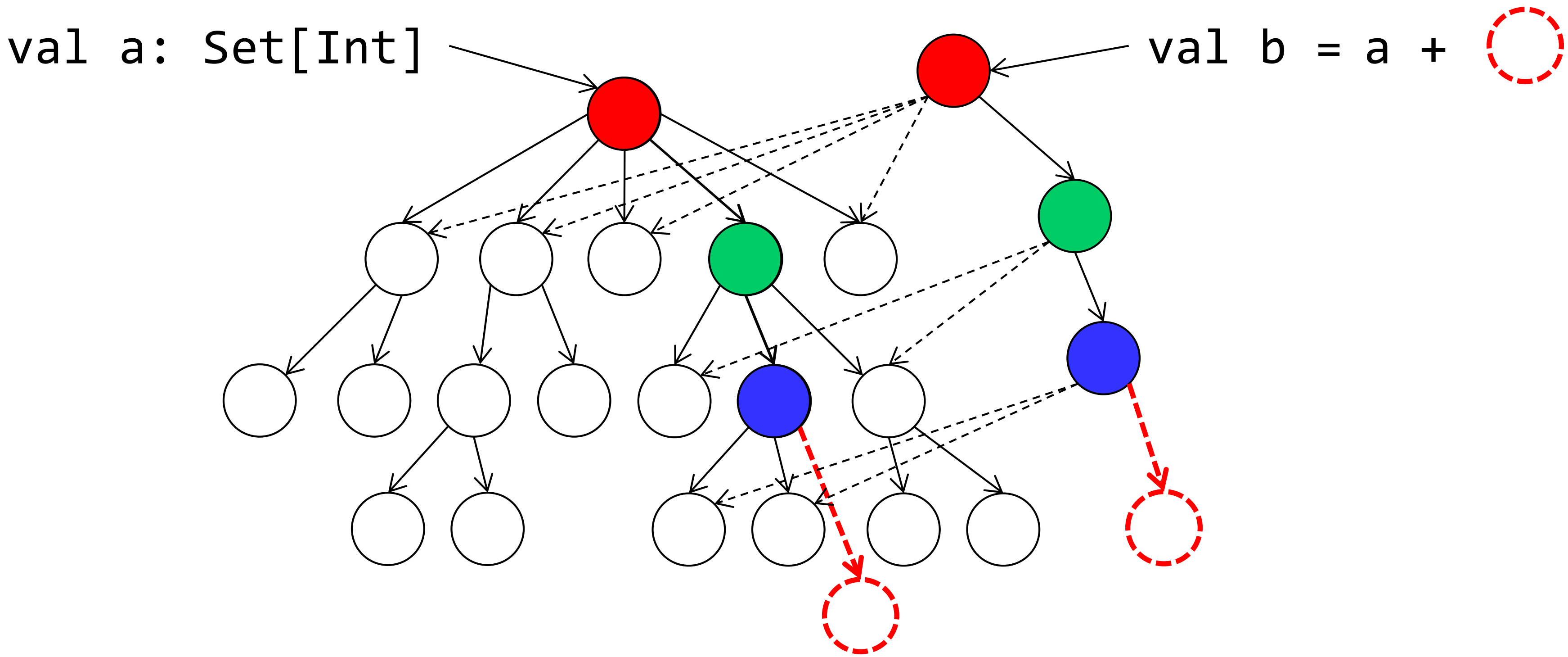
Scala Collection Overview

	mutable	immutable
concurrent	efficiently handles concurrent modifications	
parallel	parallel execution of in-place modifications	parallel execution of transformations*
sequential	sequential execution of in-place modifications	sequential execution of transformations*

Infos: https://docs.scala-lang.org/overviews/collections-2.13/introduction.html

*** Transformations like map, filter, reduce**

Immutable Datastructures



- **Operations do not modify a structure in-place but yield a new updated structure**
- **Structural sharing (path copying) makes "copies" cheap**
- **Safe to share among threads and iteration-safe**

scala.collection.immutable.List

- List is a concrete class, not an interfaces
- A list is implemented as linked list and may contain an arbitrary number of elements

```
scala> val list = List("Hello", "World", "!")
list: List[String] = List(Hello, World, !)

scala> list.head
res1: String = Hello

scala> list.tail
res2: List[String] = List(World, !)

scala> list(2)
res3: String = !

scala>  ">" :: list
res4: List[String] = List(>, Hello, World, !)
```

Selected Operations on Lists

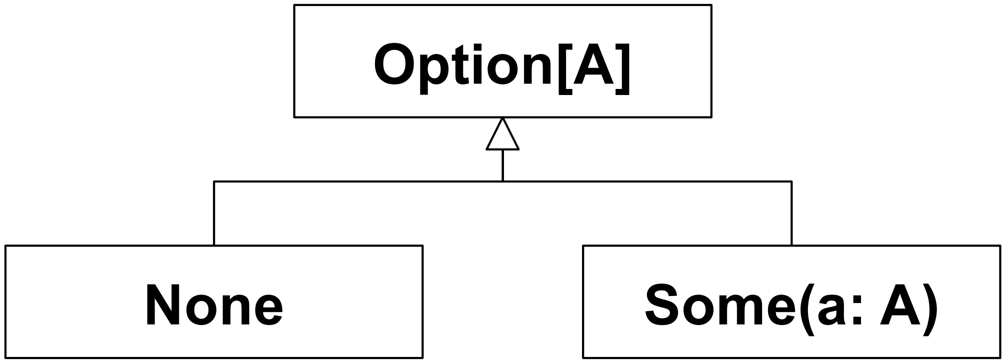
```
scala> List(1,2,3).map(i => i + 1)
res1: List [Int] = List(2, 3, 4)

scala> List(1,2,3,4).filter(i => i > 2)
res2: List [Int] = List(3, 4)

scala> List(1,2,3,4).reduce((x,y) => x+y)
res3: Int = 10

scala> List(1,2,3).zip(List('A', 'B', 'C'))
res4: List [(Int, Char)] = List((1,A), (2,B), (3,C))

scala> List("Mo", "Di", "Mi").groupBy(d => d.charAt(0))
res5: scala.collection.immutable.Map[Char,Iterable[String]]
                                = Map(D -> List(Di), M -> List(Mo, Mi))

scala> List("Mo", "Di", "Mi").find(d => d.startsWith("S"))
res6: Option[String] = None
```

- http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List
- http://docs.scala-lang.org/overviews/collections/trait-iterable.html

Excursion: Option

`class Optional<T>` (Java 8)

- **Option denotes a value which is optionally available**
  - Much more explicit than Java‘s null



- **Typically processed with pattern matching**

```
val result = List(1,2,3).find(i => i >= 2) match {
  case None => "Not Found!"
  case Some(elem) => "Found: " + elem
}
```

Fachhochschule Nordwestschweiz

java.util.Optional

```
List<String> l = Arrays.asList("Haskell", "Scala", "Java");
Optional<String> best = l.stream()
                                  .filter(s -> s.length() > 5)
                                  .findFirst();

Optional<String> upperBest = best.map(s -> s.toUpperCase());

// Either provide compensational value
String result = upperBest.orElse("No Result");

// Or execute code only if value is present
upperBest.ifPresent(s -> System.out.println(s));
```

- **Optional<T> designates a potentially absent value of type T**
  - Same as `Option[T]` in Scala and `Maybe a` in Haskell

java.util.streams.Stream

```
Stream<Integer> a = Stream.of(1,2,3).map(i -> i + 1);

Stream<Integer> b = Stream.of(1,2,3).filter(i -> i > 2);

Optional<Integer> sum = Stream.of(1,2,3,4).reduce((x,y) -> x+y);

Map<Character,List<String>> map =
   Stream.of("Mo", "Di", "Mi")
   .collect(Collectors.groupingBy(d -> d.charAt(0)));

Optional<String> day =
   Stream.of("Mo", "Di", "Mi").filter(d -> "Do".equals(d)).findFirst();
```

```
List<Integer> is = Arrays.asList(1,2,3);

is.stream().streamOp().collect(Collectors.toList())
```

- https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html

Scala Collection Overview

	mutable	immutable
concurrent	efficiently handles concurrent modifications	
parallel	parallel execution of in-place modifications	parallel execution of transformations*
sequential	sequential execution of in-place modifications	sequential execution of transformations*

Infos: http://docs.scala-lang.org/overviews/collections/introduction.html

*** Transformations like map, filter, reduce**

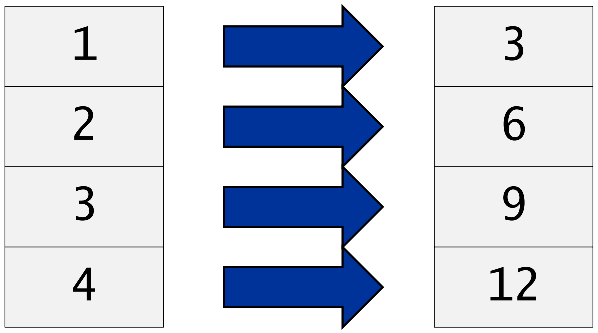
Fachhochschule Nordwestschweiz

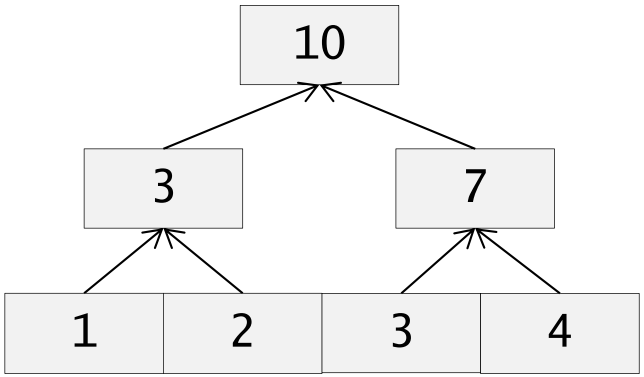
Common Operations on collections

- Common higher order functions for collection processing

```
List(1,2,3,4).map(i => i * 3)             ~> List(3,6,9,12)
List(1,2,3,4).filter(i => i % 2 == 0) ~> List(2,4)
List(1,2,3,4).reduce((i,j) => i + j)  ~> 10
```

- Observation: Those operations may safely be executed in parallel
  - Example map   reduce





Parallel Map (Exercise 9)

- **My solution**

```
def parMap[A, B](l: List[A], f: A => B): List[B] = {
  val ex = Executors.newFixedThreadPool(
                        Runtime.getRuntime().availableProcessors())

  val futures: List[Future[B]] = l.map(a => ex.submit(() => f(a)))
  val result: List[B] = futures.map(f => f.get)
  ex.shutdown()
  result
}

val result = parMap(List(1,2,3), i => i*2) //usage
```

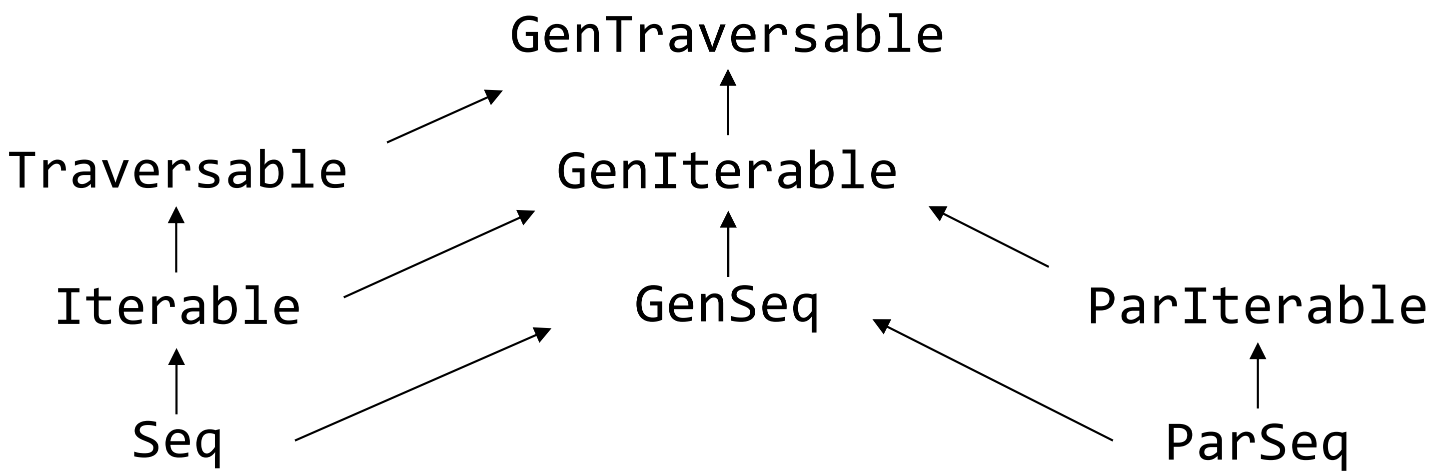
Callable

- **Their solution**

```
List(1,2,3).par.map(i => i*2).seq.toList
```

Parallel Collections

```
Stream.of(1,2,3).parallel()
 .map(i -> i + 1).sequential()
```




```
import scala.collection.parallel.CollectionConverters._
```

sequentialSeq.par →

`val sequentialSeq: Seq[Int]`   `val parallelSeq: ParSeq[Int]`

← parallelSeq.seq

http://docs.scala-lang.org/overviews/parallel-collections/overview.html

Parallel Collections Hazards

- **Side-effecting operations can lead to non-determinism**

```
scala> (1 to 5).foreach(print)       ~> 12345
scala> (1 to 5).par.foreach(print)   ~> 34512
scala> var i = 0
scala> (1 to 5).par.foreach(j => i += j) ~> i = 7
```

- **Non-associative operations lead to non-determinism**

```
scala> val p = (1 to 1000).par
scala> p.reduce(_ - _)                     ~> -228888
scala> p.reduce(_ - _)                     ~> -330101
```

- **Parallel collections are NOT concurrent collections**
- **On small collections, setup cost may be higher than performance gain**

Concurrent Programming in Scala

- **Advantages of Functional Programming**
- **Collections**
  - Immutable
  - Parallel
- **Composable Futures**
- **Reactive Programming**
  - Observables

Fachhochschule Nordwestschweiz

Refresher: Java Futures

- **Callable: Task with a result / exception**

```
interface Callable<V> {
   V call() throws Exception;
}
```

- **Submitting tasks**

```
interface ExecutorService extends Executor {
   <V> Future<V> submit(Callable<V> task);
   ...
}
```

- **Future: represents a future result of a task**

```
interface Future<V> {
  V get() throws InterruptedException, ExecutionException,
                                       CancellationException;

  ...
}
```

The problem with Java Futures

```
public class TheProblem {

  public Future[String] loadHomePage() { ... }

  public Map<String,Integer> indexContent(String content) { ... }

  public void work() throws Exception {
    // Block current Thread until result is available
    String content = loadHomePage().get();
    Map<String, Integer> index = indexContent(content);
    System.out.println(index);
  }
}
```

The Solution with Scala Futures

```
object TheSolution extends App {

  def loadHomePage(): Future[String]  = ...

  def indexContent(content: String): Map[String,Int] = ...

  // Register a callback to get notified when the result is ready
  loadHomePage().onComplete {
    case Success(m) =>
      val index = indexContent(m)
      println(index)
    case Failure(e) =>
      println(e)
  }
}
```

Scala Future Creation

- **Futures are created using the Future.apply() factory method**

```
object Future {
  def apply[T](task: => T)(implicit ec: ExecutionContext): Future[T]
}
```

Block of code to be executed asynchronously

Thread pool to execute tasks

Immediately returns a Future

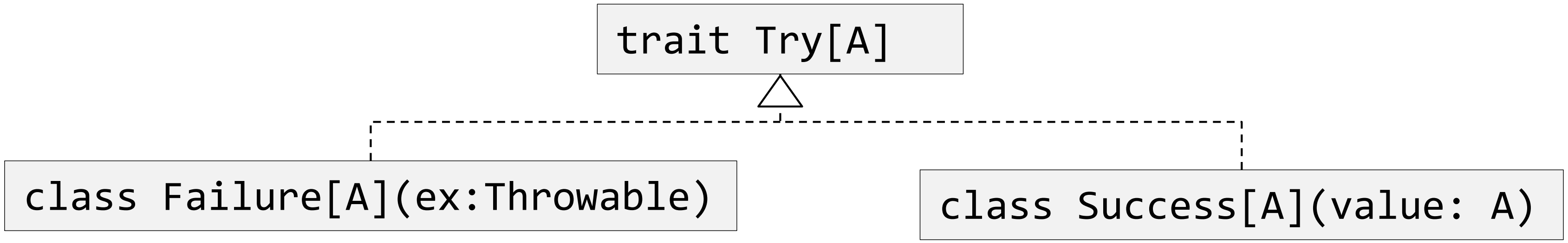
- **Usage**

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def loadHeavyData(): String = ...

val f: Future[String] = Future { loadHeavyData() }( )
```

Callbacks

- **Future allows to register completion handlers**

```
trait Future[A] {
  def onComplete[U](f: Try[A] => U]): Unit
}
```



- **Example usage**

```
future.onComplete {
  case Success(result)    => handleResult(result)
  case Failure(throwable) => handleProblem(throwable)
}
```

Fachhochschule Nordwestschweiz

Composing Futures without Blocking

- **Future[A].map(f: A => B): Future[B]**

```
def getLinks(html: String): List[URL]

val f0: Future[String] = loadHomePage()
val f1: Future[List[URL]] = f0.map(html => getLinks(html))
```

- **Future[A].flatMap(f: A => Future[B]): Future[B]**

```
def getLinksF(html: String): Future[List[URL]]

val f0: Future[String] = loadHomePage()
val f1: Future[List[URL]] = f0.flatMap(html => getLinksF(html))
```

Composing Futures

- **Future.firstCompletedOf[A](l: List[Future[A]]): Future[A]***

```
val futures: List[Future[Int]] = ...
val first: Future[Int] = Future.firstCompletedOf(futures)
```

- **Future.sequence[A](l: List[Future[A]]): Future[List[A]]***

```
val htmls: List[Future[String]] = ...
val future: Future[List[String]] = Future.sequence(htmls)
```

* The signature of the actual function is more generic

Fachhochschule Nordwestschweiz

Waiting for Results

- **Blocking until result is ready**
  - Similar to j.u.c.Future#get()

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.concurrent.duration._

val homepage = Future { loadURL("http://www.scala-lang.org/") }

// Waiting with timeout
val result = Await.result(homepage, 1 second)

// Waiting unlimited
val result = Await.result(homepage, Duration.Inf)
```

- **Blocking should be your last resort**
  - Remember the asynchronous programming mantra: "Never Block!"

j.u.c.CompletableFuture

```
public List<String> extractLinks(String http) { ... }
public String loadHomePage(String url) {...}

// runs loadHomePage on ForkJoinPool.commonPool()
CompletableFuture<String> html =
                CompletableFuture.supplyAsync(() -> loadHomePage());

// execute extractLinks in the same thread (onSuccess Callback)
CompletableFuture<List<String>> links1 =
                         html.thenApply(s -> extractLinks(s));

// block until result is ready
List<String> list = links1.get();
```

Concurrent Programming in Scala

- **Advantages of Functional Programming**
- **Collections**
  - Immutable
  - Parallel
- **Composable Futures**
- **Reactive Programming**
  - Observables

Fachhochschule Nordwestschweiz

Reactive Programming with RxScala

	Sync	Async
Single	getData(): **T**	getData(): **Future[T]**
Multiple	getData(): **Iterable[T]**	getData(): **Observable[T]**

Iterable

- pull model
- Sequence of elements
  - block until available
  - `val e = i.next()`

Observable

- push model
- Sequence of events
  - get notified as they happen
  - `onNext(e)`

Reactive Programming with RxScala

- **RxScala is a library for composing asynchronous and event-based programs using observable sequences.** http://reactivex.io/rxscala/
- **Examples for event based programs**
  - Mouseevents in a GUI application
  - Datafeeds: Sensor data, stock exchange feeds
  - Networking: Wikipedia edits IRC, JMS based applications
- **Implementations for many languages are available**
  - Original implementations from Microsoft for C# and JavaScript
  - Many adapters for JVM based languages

Iterable vs Observable

Iterable	Observable
pull	**push**
`next(): T`	`onNext(t: T)`
`throws Exception`	`onError(t: Throwable)`
`hasNext() == false`	`onCompleted()`

```
// Iterable[String]
// containing some HTML strings
getDataFromLocalMemory()
  .drop(7)
  .filter(s => s.startsWith("h"))
  .take(12)
  .map(s => toJson(s))
  .foreach(j => println(j))
```

```
// Observable[String]
// emitting some HTML strings
getDataFromNetwork()
  .drop(7)
  .filter(s => s.startsWith("h"))
  .take(12)
  .map(s => toJson(s))
  .subscribe(j => println(j))
```

Reactive Programming with RxScala

- **Trait Observable represents an observable sequence of events**

```
trait Observable[T] {
  def subscribe(obs: Observer[T]): Subscription
}
```

- **Observers can be subscribed to be notified when events occur**

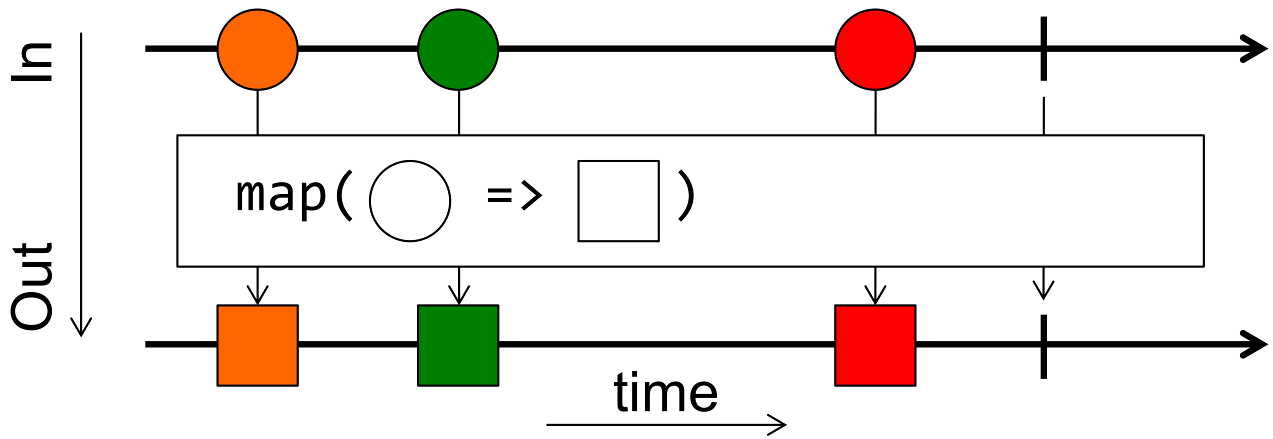
```
trait Observer[T] {
  def onNext(t: T): Unit
  def onCompleted(): Unit
  def onError(t: Throwable): Unit
}
```

- **A Subscription can be cancelled**

```
trait Subscription {
  def unsubscribe(): Unit
  def isUnsubscribed(): Boolean
}
```

Observable Example

- **Contract:** `onNext* (onComplete | onError)?`
  - onNext is called multiple times (zero times is possible) followed by either onComplete of onError.
- **Marble diagrams**



Conclusion

- **Concurrency Primitives**
  - Available, but seldom a clever choice
- **Collections**
  - Immutable by default, Rich API, simple to use, gateway drug
- **Parallel collections**
  - Just append **.par** and get all available compute power of your system
- **Composable futures**
  - Support nonblocking asynchronous programming (“never block!”)
- **Reactive Programming with RxScala**
  - Composing event-based programs using observable collections
- **This is only the beginning**
  - STM, transactional heap access
  - Actors, share nothing concurrency

Concurrent Programming in Scala

- **Concurrency Primitives**
  - References
  - volatile
  - synchronized

References

- **val**
  - Same semantics as final (initialization guarantees)

```
scala> val v = 42; v = 13
<console>:8: error: reassignment to val
       val v = 42; v = 13
                     ^
```

- **lazy val**
  - Lazy initialization / threadsafe

```
scala> lazy val l = { println("init"); 42 }
l: Int = <lazy>
scala> l
init
res0: Int = 42
scala> l
res1: Int = 42
```

References

- **var**
  - Mutable / Same as ordinary variable in Java

```
scala> var a = 42
a: Int = 42

scala> a = 13
a: Int = 13
```

- **@volatile var**
  - Volatile is implemented as annotation

```
scala> @volatile var a = 42
a: Int = 42

scala> a = 13
a: Int = 13
```

synchronized

- **Implemented as a method on AnyRef (same as Object in Java)**

```
def synchronized[T](block: => T): T
```

- **Simple usage**

```
val lock = new Object()
lock.synchronized {
  // Region guarded by lock
}
```

```
Object lock = new Object();
synchronized(lock) {
  // Region guarded by lock
}
```

- **Guarded block can return a result**

```
val lock = new Object()
val result = lock.synchronized {
  // Region guarded by lock
  "Return value"
}
```

```
Object lock = new Object();
String result = null;
synchronized(lock) {
  // Region guarded by lock
  result = "Return value"
}
```