RECAP Functions and Pattern matching!

```
trait Function1[-A, +R] {
  def apply(x: A): R
}
```

The pattern matching block...

```
{ case (key, value) => key + ": " + value }
```

expands to the Function1 instance

```
new Function1[JBinding, String] {
  def apply(x: JBinding) = x match {
    case (key, value) => key + ": " + value.toString
  }
}
```

```
{ case "ping" => "pong" }
```

can be given type String => String

We need to give it a type from outside

```
val F: String => String = { case "ping" => "pong" }
```

F("ping") we get "pong"

F("abc") we get match error

We can make it a partial function

```
val F: PartialFunction[String, String] =
  { case "ping" => "pong" }
```

F.isDefinedAt("ping") true

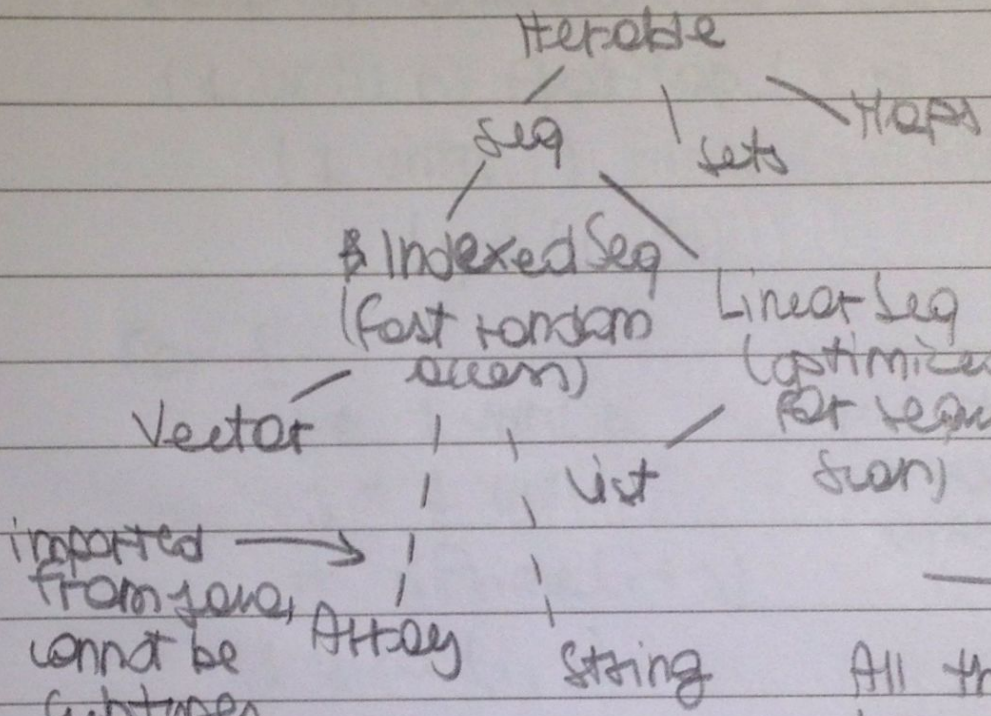
A partial function is defined as follows

```
trait PartialFunction[-A, +R] extends Function1[-A, +R] {
    def apply(x: A): R
    def isDefinedAt(x: A): Boolean
}
```

{ case "ping" => "pong" } extended to...

```
new PartialFunction[String, String] {
    def apply - ...
    def isDefinedAt(x: String) = x match {
        case "ping" => true
        case _ => false
```

RECAP Collections



All those types, all share a common set of general methods

map
flatMap > core
Filter

Fold Left
Fold Right

```
def map[U]
(F: T=> U): List[U] =
this match { case x :: xs => F(x) :: xs.map(F)
             case Nil => Nil
}}

def flatMap[U](f: T=>List[U]): List[U]=
this match { case x :: xs => F(x) ++ xs.flatMap(F)
             case Nil => Nil

def filter(p: T => Boolean): List[T]=this match{
  case x :: xs => if (p(x)) x :: xs filter p
  case Nil => Nil }}     else xs filter p
```

For expressions

Simplify combinations of core methods

```
(1 until n) flatMap (i =>
    (1 until i) filter (j => isPrime(i+j)) map
        (j => (i, j)))
```

```
for {
    i ← 1 until n          yield the pairs
    j ← 1 until i          in range 1 to n
    if isPrime(i+j)        whose sum
} yield (i, j)             is prime
```

==

Here's how the scala compiler translates for expressions

1. A simple for-expression

   `for (x ← e1) yield e2`

   becomes `e1.map(x => e2)`

2. For-expression with a filter

   `for(x ← e1 if F; s) yield e2`

   where F is a filter and s is a sequence of generators and filters ...

becomes `for(x ← e1.withFilter(x => F); s) yield e2`

withFilter can be thought of as a variant of filter that doesn't produce an intermediate list, but instead filter, map & flatmap function application

3. With two generators in sequence

for (x <- e1; y <- e2; s) yield e3

where s is a sequence of generators and filters

becomes e1.flatMap(x => for (y <- e2; s) yield e3)

—

Each of these rules translates to for-expression with one fewer element between the parentheses:

1 eliminates a single generator
2 eliminates a filter
3 eliminates the leading generator

The lhs of a generator in a for expression can also be a pattern.

They'll get [redacted] as implicit filters

added

map, flatMap, filter can be defined in terms of for

```
def mapFun[T,V](xs: List[T], f: T => Iterable[V])
  : List[V] = for(x <- xs) yield f(x)

def flatMap[T,V](xs: List[T], f: T => Iterable[V])
  : List[V] = for(x <- xs; y <- f(x)) yield y

def filter[T](xs: List[T], p: T => Boolean): List[T] =
  for(x <- xs if p(x)) yield x
```

In reality, the scala compiler translates for expressions to combinations of map, flatMap and a variant of filter (lazy)

```
for(x <- e1) yield e2
  e1.map(x => e2)        1 Simple for expr!

for(x <- e1 if f; s) yield e2
```

↓ translated to:

```
for(x <- e1.withFilter(x => f); s) yield e2
```

↑ more filters / generators

s can be empty!

↑ filter absorbed by generator (generator reduced to elements that pass condition f)

2. for expr where f is a filter and s is a sequence of generators:

```
for(x <- e1; y <- e2; s) yield e3
```

↓ translated to

```
e1.flatMap(x => for(y <- e2; s) yield e3)
```

3. A for expr where s is a sequence of generators and filters is ~~translated to~~

1 → Direct translation into a map

2/3 → for expression to another with one less element

It will then translate into a map once we hit 1 after 2/3 simplifications.

—

For expr are not limited to collections. They can be used with any type with map, withFilter, FlatMap defined

Functional Random generators:

Get random values for other domains

```
trait Generator[+T] {
  def generate: T
}

val integers = new Generator[Int] {
  val rand = new java.util.Random
  def generate = rand.nextInt()
}

val booleans = new Generator[Boolean] {
  def generate = integers.generate > 0
}

val pairs = new Generator[(Int, Int)] {
  def generate = (integers.generate, integers.generate)
}
```

Ideally, we can write

val booleans = for(x ← integers) yield x > 0

def pairs[T,U](t: Generator[T], u: Gen.[U]) =
  for{x ← t; y ← u} yield (x, y)

We need to define map and flatMap

```
trait Generator[+T] {
    self =>  // an alias for this
    def generate: T
  def map[S](F: T => S): Generator[S] =
      new Generator[S] {
          def generate = F(self.generate)
      }
  def FlatMap[S](F: T => Generator[S])
  :Generator[S] = new Generator[S] {
      def generate = F(self.generate).generate
  }

def single[T](x: T): Gen[T] = new Gen[T] {
      def generate = x }
def choose(lo: Int, hi: Int): Gen[Int] =
  for (x ← integers) yield lo + x % (hi - lo)
```

List generator

```
def lists: Gen[List[Int]] = for { isEmpty ← booleans
  list ← if(isEmpty) emptyLists else nonEmptyLists
  } yield list

def emptyLists = single(Nil)
def nonEmptyLists = for { head ← integers
  tail ← lists } yield head :: tail
```

Tree generator

```
def leafs: Gen.[Leaf] = for { x ← integers
  } yield Leaf(x)

def trees: Gen.[Tree] = for { isLeaf ← booleans
  tree ← if (isLeaf) leafs else inners
  } yield tree
```

RANDOM TESTING!

Unit tests

- You come up with some test inputs to program functions and a postcondition
- The postcondition is a property of the expected result
- Verifies that the program satisfies the postcondition

There may be other test inputs on which the program could fail

We can do it without finding the test inputs if we can generate random test inputs!

```
def test[T](g: Gen[T], numTimes: Int=100)
    (test: T => Boolean): Unit {
        for (i <- 0 until numTimes) {
            val value = g.generate
            assert(test(value), "test failed
                for "+value)
        }
    print("passed "+numTimes+" tests")
}
```

=

The idea of random tests ~~generate~~ and random value generators is embodied in a tool called ScalaCheck

It would come up with random value generators, sometimes just by giving the type, and then run tests.

↙ test domains!

```
forAll { (l1: List[Int], l2: List[Int]) =>
    l1.size + l2.size == (l1++l2).size
```

If a test fails, it can minimize the counter example, ~~then~~ ~~try~~ it tries to repeatedly find smaller and smaller examples until the ~~one~~ example it is small enough to be easy to understand as a counterexample

Monads

Data structures with map and flatMap seem to be quite common.

In fact there's a name that describes this class of data structures together with some algebraic laws that they should have, called monads

A monad is a parametric type M[T] with two operations: flatMap and unit (also called bind)

```
trait M[T] {
  def flatMap[U](F: T => M[U]): M[U]
}
def unit[T](x: T): M[T]
```

List is a monad with unit List(x), same for Set, Option and generator (single(x) or unit)

flatMap is an operation in each of these, whereas unit is different for each monad — m map F == m flatMap (x => unit(F(x)))

To qualify as a monad, a type has to satisfy three laws

m flatMap F flatMap G = m flatMap (F flatMap G)

Associativity!

m flatMap (F andThen unit) = m

unit(x) flatMap F = F(x)

unit!

m flatMap unit = m

right unit!

Significance of the laws for For expr.

1 Associativity says that one can inline nested for expressions

```
For(y ∈ For(x ∈ m; y ∈ F(x)) yield y
    z ∈ G(y)) yield z   =
for (x ∈ m; y ∈ F(x); z ∈ G(y)) yield z
```

2 Right unit says `For(x ∈ m) yield x = m`

3 left unit does not have an analogue for For expr

Try is another type that resembles Option, except that there's two different cases, Success and Failure that contains an exception.
Try is used when we want to propagate exception between different threads or processes

```
abstract class Try[+T]
case class Success[T](x:T) extends Try[T]
case class Failure(ex:Exception)
                  extends Try[Nothing]
```

↑ Bottom subtype

static types!

Types allow you to denote function domains and codomains

Given types the compiler can now statically verifies that the program is sound

With increasing expressiveness in type systems we can produce more reliable code

All type information is erased at compile time

Types in Scala

- parametric polymorphism, roughly generic programming
- (local) type inference, you know that one
- existential quantification, roughly, defining something for some unnamed type
- views, roughly, "castability" of values of one type to another

You can wrap up an arbitrary computation in a try Try(expr)

Here's an implementation of Try

```
object Try {
def apply[T](expr: => T): Try[T] =
  try Success(expr) catch {
    case NonFatal(ex) => Failure(ex)}
```

expr passed by name because otherwise it wouldn't be a computation that could throw an exception!

Just like with option, try-valued computations can be composed in for expressions

```
for { x ← computeX; y ← computeY }
    yield F(x, y)
```

If computeX and computeY both succeed with success and success(y), that will return success(F(x, y)). But if either computation fails, this will return Failure(ex)

flatMap and map on Try

```
def flatMap[U](F: T ⇒ Try[U]): Try[U] =
this match { case Success(x) ⇒ Try(F(x))
             catch { case NonFatal(ex) ⇒ Failure(ex) }
             case Fail: Failure ⇒ Fail

def map[U](F: T ⇒ U): Try[U] = this match {
    case Success(x) ⇒ Try(F(x))
    case Fail: Failure ⇒ Fail }
```

This is not a monad! It could be with unit = Try(expr) but we want this to never throw a non fatal exception in a statement like Try(expr).flatMap(F(expr)) != F(expr) that fails the left unit law.

An exception with Try.map f can't be never throw a

Streams!

We've seen a number of immutable collections that provide powerful operations for combinatorial search

```
((1000 to 10000) filter isPrime)(1)
```

2nd prime between 100 and 1000

which is more elegant than

```
nthPrime(from: Int, to: Int, n: Int): Int =
  if (from >= to) throw new Error
  else if (isPrime(from))
    if (n == 1) from else (from+1,
      to, n-1)
    else nthPrime(from+1, to, n)
```

But the shorter expression is inefficient because we construct all primes between 1000 and 10000 only to the the second element!

Reducing the bound would speed up but you can't be sure whether the bound is too low and you would miss the prime number

Idea, avoid computing tail of a sequence until it is needed for evaluation result

The Stream is a class similar to lists which tail is evaluated on demand

Streams can be build from Stream.empty and a constructor Stream.cons

val xs = Stream.cons(1, Stream.cons(2, Stream.empty))

or Stream(1,2,3), like other collections

toStream will turn a collection into a stream

(1 to 1000).toStream

=> res0: Stream = Stream(1, ?)

=

Function that returns (lo until hi).to Stream

```
def listRange(lo:Int, hi:Int): List[Int] =
  if (lo>=hi) Nil
  else lo :: listRange(lo+1, hi)
```

compare to ↓

```
def streamRange(lo:Int, hi:Int): Stream[Int]
= if (lo>=hi) Stream.empty
  else Stream.cons(lo, streamRange(lo+1, hi)
```

=

Streams support all the methods of List

(1000 to 10000).toStream filter isPrime(_)1)

:: always produces a list, never a stream

#:: produces a stream! x#::xs = Stream.cons(x, xs)

↳ can be used in patterns!

Implementation of streams

```
trait Stream[+A] extends Seq[A] {
  def isEmpty; def head: A; def tail: Stream[A] }
    : Boolean
```

ONLY DIFFERENCE! *

```
object Stream {
  def cons[T](hd: T, tl: => Stream[T]) =
    new Stream[T] { def isEmpty = false
      def head = hd ; def tail = tl }

  val empty = new Stream[Nothing] {
    def isEmpty = true
    def head = Error def tail = Error }
}
```

* When I first construct a cons cell for a stream, the tail is not evaluated, it is the first time somebody dereferences it! (calls def tail!)

The other methods are implemented analogously to their list counterpart

Lazy evaluation!

The proposed implementation of streams solves the problem of avoiding unnecessary computations, but it suffers from another performance problem: If tail is called several times, the corresponding stream will be recomputed each time

Idea, store the result of the first evaluation of tail, and re use the stored result instead of recomputing it

~~The idea is a purely Functional language, call by name evaluation~~

We call this ~~the~~ where lazy evaluation (as opposed to by-name evaluation where everything is recomputed and strict evaluation for normal parameters and val definitions)

Haskell uses it by default!

Scala doesn't do it as lazy val can be unpredictable

so it uses strict evaluation by default but allows lazy evaluations with ~~lazy~~ value definitions with a "lazy val" Path

lazy val x = expt

↳ def x = expt

the not evaluated here at definition but after all its name when x is accessed

- the difference is that def is then re-evaluated each time, while lazy val the expr is remained with everytime except for the first one

[redacted] Using lazy val for tail, Stream.cons can be more efficient

```
def cons ... = new Stream {
  def head = hd
  lazy val tail = tl
```

```
def cons[T](hd: T, tl: => Stream[T]) =
  new Stream[T] {
    def head = hd
    def tail = tl
    ...
  }
```

Define infinite streams?

All ints starting from a given number

```
def From(n: Int): Stream[Int] =
    n #:: From(n+1)
```

Sieve of Eratosthenes

- start wh all integers starting from 2, prime
- Eliminate multiples of 2
- The first number of the resulting list is 3, prime
- Eliminate all multiples of 3
- Iterate. At each step, the first number in the list is prime and eliminate its multiples

```
def sieve(s: Stream[Int]): Stream[Int] =
    s.head #:: sieve(s.tail filter (_ % s.head != 0))

val primes = sieve(From(2))
(primes take(100)).toList
```

Square roots

Our previous square root algorithm used isGoodEnough to tell when to terminate the iteration.

With streams we express the concept of a converging series without having to worry about when to terminate it

...

```
def sqrtStream(x: Double): Stream[Double] = {
  def improve(guess: Double) =
    (guess + x / guess) / 2
  lazy val guesses: Stream[Double] =
    1 #:: (guesses map improve)
  guesses
}
```

=

```
val xs = from(1) map (_ * N)
val xs = from(1) filter (_ % N == 0)
```

map does not generate unnecessary stream elements and therefore it's more efficient

Functions and state

Until now our programs have been side effect free

For all programs that terminate any sequence of evals give the same result

...

```
if (1==0) 3 else iterate(1-1, square, square(3))
```

```
iterate(0, square, square(3))        if(i==0) 3 else iterate(i-1, square, 3*3)
```

g

Confluence, discovered by Church and Russell, also called Church-Russell theorem

that holds for pure functional programming. But we now want to introduce state

Some objects have state that changes over the course of time

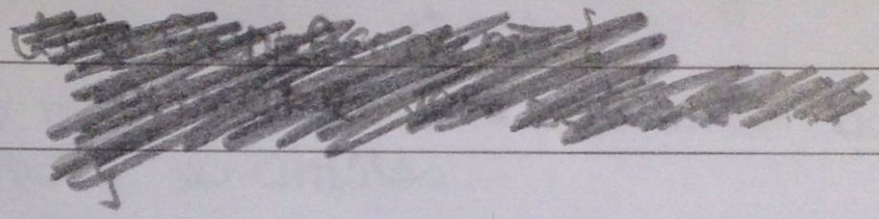
"An object has state if its behaviour is influenced by its history"

Every form of mutable state is constructed from some variables

To define a variable use var!

A variable can be reassigned

Objects with state are usually represented by objects that have some variable members



Identity and change

val x = E ; val y = E E → expression

then x and y are the same

val x = E ; val y = x is same as above

operational equivalence defines the property of being the same

suppose you have x and y, they are operational equivalent if no possible test can distinguish between them

val x = new ...
val y = new ... ← F is every possible test

F(x,y) F(x,x)

IF F(x,y) is different from F(x,x) the objects are different

val x = new BankAcc val x = new BankAcc
val y = new BankAcc val y = new BankAcc
x deposit 30 x deposit 30
y withdraw 20 x withdraw 20

the final results are different, they're not equal!

```
val x = new BankAcc
val y = x
```

* NO sequence of operations can distinguish between them

The model of substitution cannot be used in those examples.

According to this model, one can always replace the name of a value by the expression that defines it. For example

```
val x = new BankAcc        val x = new BankAcc
val y = x          ⟶       val y = new BankAcc
```

the x in the definition of y could be replaced by new BankAcc, but we've seen that this change leads to a different program.

The substitution model is not valid with assignments

It is possible to adapt the substitution model, but that becomes more complicated

Variables are enough to model all imperative programs

Loops are not essential for imperative languages, and can be modelled using functions

```
def WHILE (condition: => Boolean)(command: => Unit)
: Unit = if (condition) {
           command
           WHILE (condition)(command)
         } else ()
```

- conditions are passed by value so that they are reevaluated in each iteration
- WHILE is tail recursive

```
def REPEAT (command: => Unit)(condition: => Boolean = {
    command
    if (condition) ()
    else REPEAT (command) (condition)
}
```

The classical For loop cannot be modeled simply by a higher order function

Scala has a similar kind of For loop

```
For (i <- 1 until 3) print(i)
```

For loops translate quite similarly to For expressions, but For expressions translate into combinations of map and flatmap, For loops into combinations of a function foreach

```
Foreach (F: T => Unit): Unit =
```

// apply f to each element of the collection

for (i ← 1 until 3; j ← "abc") println(i+" "+j)

(1 until 3) foreach (i ⇒ "abc" foreach (j ⇒ println (i+" "+j))

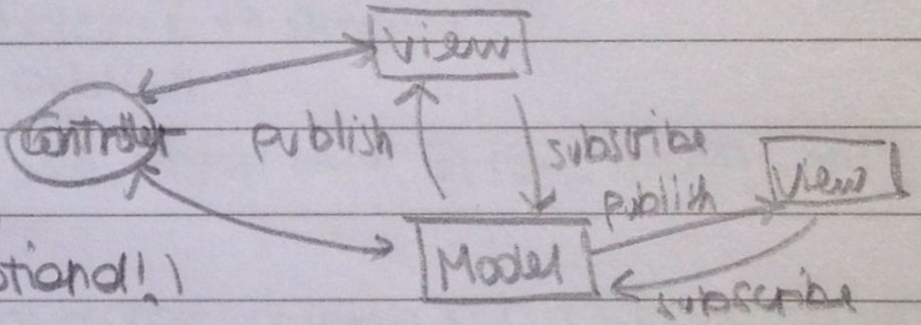
Timely effects =

The observer pattern is widely used when views need to react to changes in a model

present the properties of the model in some way

contains the state of an application

manages interactions between view and model (optional!)




```
trait Publisher {
  private var subscribers: Set[Subscriber] =
  def subscribe(subscriber: Subscriber) {
    subscribers += subscriber

  def unsubscribe(subscriber: Subscriber =
      subscribers -= subscriber

  def publish() = subscribers.foreach(_.handler
                                      (this))

trait Subscriber {
  def handler(pub: Publisher)
}
```

```
class BankAccount extends Publisher {
  private var balance = 0
  def currentBalance = balance
  def deposit(amount: Int) =
    if (amount > 0) {
      balance += amount
      publish }

  def withdraw(amount: Int) =
    if (0 < amount && amount <= balance) {
      balance -= amount
      publish }
    else throw new Error("Insufficient
                          funds")
}

class Consolidator(observed: List[BankAccount])
  extends Subscriber {
  observed.foreach(_.subscribe(this))
  private var total: Int = _
  compute()

  private def compute() =
    total = observed.map(_.currentBalance)
                    .sum

  def handler(pub: Publisher) =
    compute()

  def totalBalance = total
}
```

Observer pattern

Observer pattern, The good

- Decouples view from state
- Varying number of views for a given state
- Simple to set up

The bad

publish, subscribe handle

- Forces imperative style, since handlers return Unit (so they must be imperative)
- Many moving parts that need to be coordinated
- Concurrency makes things more complicated
- Views are still tightly bound to the state, view updates happens immediately

Functional Reactive Programming

Reactive programming is all about reacting to sequences of events that happen in time

Functional view: Aggregate an event sequence into a signal

- A signal is a value that changes over time
- Represented as a function from the time domain to the value domain
- Instead of propagating updates to mutable state, we define new signals in terms of existing ones

Example: Mouse positions

Event-based view:

Whenever the mouse moves, an event
MouseMoved(toPos: Position)
... is fired I will update current state (imperative)

FRP view:

A signal mousePosition: Signal[Position]
which at any point in time represents
the current mouse position

Origins of FRP

started in 1997 with the paper Functional Reactive Animation by Elliott, Hudak

There have been many FRP systems since

We will introduce FRP with a minimal class Frp.Signal, which we will define ourselves

Frp.Signal is modeled after scala.react, described in the paper Deprecating the Observer Pattern

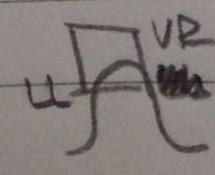
Fundamental Signal Operations

- Obtain the value of a signal at the current time.
  In our library it is expressed by ()

e.g. mousePosition() // the current mouse position

- Define a signal in terms of other signals
  In our library expressed by the Signal constructor

```
def inRectangle(LL: Position, UR: Position): Signal[Boolean] =
  Signal { val pos = mousePosition()
           LL <= pos && pos <= UR }
```



Constant Signals

The Signal(3) syntax can also be used to define a signal that let always the same value

```
val sig = Signal(3)
```

Time-Varying Signals

- We can use externally defined signals (such as mouseposition) and map over them to have new time-varying signals
- Or we can use a Var (subtype of Signal)

=

Variable Signals

Values of type Signal are immutable

But our library define a subtype Var of Signal that can changed

Var provides an update operation, which allows to redefine the value of a signal from the current time on

```
val sig = Var(3)
sig.update(5) // From now on, sig is 5
```

In scala, calls to update can be written as assignments

```
sig() = 0   really is   sig.update(i, 0)
class Array[T] { def update(idx: Int, value: T) = ... }
```

Indexed assignment like $f(E_1, \ldots, E_n) = E$
is translated to $f.update(E_1, \ldots, E_n, E)$

For $n=0$, $F()=E$ is shorthand for $F.update(E)$

So sig.update(5) really is sig() = 5

Vars look like mutable variables, but there's a crucial difference

We can measure signals, which gives us a relation between two signals that is maintained automatically at all future points in time

```
a = 2
b = 2 * a
a = a + 1
```
↑ b is not updated!

```
a() = 2
b() = 2 * a()
a() = 3
```
↑ b is updated!

```
class BankAccount {
  private val balance = Var(0)
  def deposit(amount: Int) =
    if (amount > 0)
      balance() = b - amount
  def withdraw(amount: Int) = balance()
    if (amount > 0 && amount <= b) {
      balance() = amount + b
    } else throw new Error("Insufficient funds")
}
def consolidated(accts: List[BankAccount]
  : Signal[Int] =
    Signal(accts.map(_.balance()).sum)
```

val b = balance()

val b = balance()

```
def exchange = Signal(246.00)   ← Add exchange dollar with bitcoins
val inDollar = Signal(c() * xchange())
```

A Simple FRP Implementation

```
class Signal[T](expr: => T) {
  def apply(): T = ???
}

object Signal {
  def apply[T](expr: => T) = new Signal(expr)
}

class Var[T](expr: => T) extends Signal[T](expr) {
  def update(expr: => T): Unit = ???
}

object Var {
  def apply[T](expr: => T) = new Var(expr)
}
```

Each signals maintains
- The current value
- The expression that defines the signal
- A set of observers: other signals that depend on the value of the current signal

If signal changes, all observers need to be re-evaluated

How do we record dependencies in observers?

- When evaluating a signal-valued expression, we need to know which signal caller gets defined or updated by the expression
- If we know that, then executing sig() means adding a caller to the observers of sig
- When signal changes, all previously observing signals are re-evaluated and the observers in sig are cleared
- Re-evaluation will re-enter a calling signal caller in sig.observers, as long as caller's value still depends on sig

How do we find out on who's behalf a signal expression is evaluated

One simple way to do this is to maintain a global structure referring to the current caller

That data structure is accessed in a stack-like fashion, because one evaluation of a signal might trigger others

```
class StackableVariable[T](init: T) {
  private var values: List[T] = List(init)
  def value: T = values.head
  def withValue[R](newValue: T)(op: => R): R =
  { values = newValue :: values
    try op finally values = values.tail
  }
}
object NoSignal extends Signal[Nothing](???) { ... }

object Signal {
  private val caller = new StackableVariable
  def apply[T](expr: => T) = [Signal[_]](NoSignal)
    new Signal(expr)
}

class Signal[T](expr: => T) {
  import Signal._
  private var myExpr: () => T = _
  private var myValue: T = _
  private var observers: Set[Signal[_]] = Set()
  update(expr)
  protected def update(expr: => T): Unit = {
    myExpr = () => expr
    computeValue()
  }
```

```
protected def computeValue(): Unit = {
  myValue = caller.withValue(this)(myExpr())
}

def apply() = {
  observers += caller.value
  assert(!caller.value.observers.contains(this), "cyclic signal definition")
  myValue
}
} ← close Signal def
```

A signal's current value can change when
- somebody calls an update operation on a var or
- the value of a dependent signal changes

propagating requires a more refined implementation of computeValue

```
protected def computeValue: Unit = {
  val newValue = caller.withValue(this)(myExpr())
  if (myValue != newValue) {
    myValue = newValue
    val obs = observers
    observers = Set()
    obs.foreach(_.computeValue())
  }
}

object NoSignal extends Signal[Nothing](...) {
  override def computeValue() = ()
}
```

↗ disable computeValue for NoSignal

all functionalities for Var is already present in Signal

```
class Var[T](expr: => T) extends Signal[T](expr) {
  override def update(expr: => T): Unit =
    super.[redacted](expr)
         update
}
```

override update to make it public (was protected)

```
object Var {
  def apply[T](expr: => T) = new Var(expr)
}
```

[redacted]

One way to get around the problem of concurrent accesses to global state is to use synchronization

But this blocks threads, can be slow and can lead to deadlocks

Another solution is to replace global state by a thread local state

- Thread local state means that each thread accesses a separate copy of a variable
- It is supported by scala through scala.util.DynamicVariable

```
object Signal {
  private val caller = new DynamicVariable[
      Signal[_]](NoSignal)
  [redacted]
}
```

↑ instead of StackableVariable

Thread-local states have a number of disadvantages

- It is imperative nature, often produces dependencies which are hard to manage
  - Its implementation on the JDK involves a hashTable lookup which can be a performance problem
- It does not play well in situations where threads are multiplexed between several tasks

Another solution, Implicit parameters

- Instead of maintaining a thread-local variable, we pass its current value into a block expression as an implicit parameter
  - this is purely functional. But it currently requires more boilerplate than the thread-local solution
  - Future versions of scala might solve that problem

Latency as an effect

"Computation can take time"

```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 50,000 ns
val confirmation = socket.sendToEurope()
// block for 150,000,000 ns
```

Both blockings could throw an exception

~~callbacks~~, if a computation takes a lot of time, we introduce a callback that will be called when the computation terminates either ~~successfully or with an exception~~

=

Future[T] is a monad that handles both ~~exceptions~~ and latency

~~import ...~~

```
trait Future[T] { ...
  def onComplete(callback: Try[T] => Unit)
      (implicit executor: ExecutionContext)
    }                                    : Unit
```

You can pattern match it like

```
ts match {
  case Success(t) => onNext(t)
  case Failure(e) => onError(e)
}
```

```
trait Socket {
  def readFromMemory(): Future[Array[Byte]]
  def sendToEurope(packet: Array[Byte]):
                                  Future[Array[Byte]]
}

val socket = Socket()
val packet = Future[Array[Byte]] =
    socket.readFromMemory

packet.onComplete {
    case Success(p) => {
        val confirmation: Future[Array[Byte]] =
            socket.sendToEurope(p)
    }
    case Failure(t) => ...
}
                                // Not great...
```

// Starts an asynchronous computation and returns a Future object to which you can subscribe to be notified when the future completes

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

combinators on Future:

(flatMap, map, filter...)

Future has common higher-order operators and specific ones such as recoverWith

// Better!

```
val socket = Socket()
val packet: Future[Array[Byte]] = socket.readFromMemory()
val confirmation: Future[Array[Byte]] =
  packet.flatMap(p => socket.sendTO(europe)(p))
```

=

```
def recover(f: PartialFunction[Throwable, T]): Future[T]
def recoverWith(F: PartialFunction[Throwable, Future[T]]): Future[T]
```

↑ flatMap for Error case

Map for the error case

```
def fallbackTo(that: => Future[T]): Future[T] = {
  this recoverWith {
    case _ => that recoverWith {case _ => this}
  }
}
```

↑ beautiful ☺

```
def sendToSafe(packet: Array[Byte]): Future[Array[Byte]] =
  sendTO(mailServer.europe, packet)
    .fallbackTo {sendTO(mailServer.usa, packet)} mapTo {
      case t: Throwable => noEmail.getMessage.toByteArray
    }
```

Sometimes you want to block (like printing), there's Awaitable that allows to block on a Future with a timeout duration.

~~Use only in main, not in production code~~

```
val socket = Socket()
val packet: Future[Array[Byte]] =
    socket.receiveFromMemory()
val confirmation: Future[Array[Byte]] =
    packet.flatMap(socket.sendToSafe(_))
val c = Await.result(confirmation, 2 seconds)
println(c.toNext)
```

You always want asynchronous and never block! except for little scripts.
(You might as well not use asynchrony

postfix operator
(3 minutes wh
import scala.
language.postfixOps)

composing futures

We can rewrite the code x<- wh for comprehension like

```
val socket = Socket()
val confirmation: Future[Array[Byte]] = for {
    packet <- socket.receiveFromMemory()
    confirmation <- socket.sendToSafe(packet)
} yield confirmation
```

```
def retry(noTimes: Int)(block: => Future[T]):
    if (noTimes == 0) Future.failed(
                        new Exception("sorry"))
    else block.fallbackTo {
        retry(noTimes-1){block}
    }
}
}
```

Future[T] ={

Closer look at FlatMap in terms of onComplete:

```
trait Future[T] {
  def onComplete(callback: Try[T] => Unit) = ...
  def FlatMap[S](F: T => Future[S]): Future[S] = ???
}
```

```
def flatMap... = new Future[S] {
  def onComplete(callback: Try[S] => Unit): Unit =
    self onComplete {
      case Success(x) => F(x) onComplete(callback)
      case Failure(e) => callback(Failure(e))
    }
}
```

=

"When we do thing wh recursion, is often nice to do it wh a fold"

List(a,b,c).foldRight(e)(F) ← Function
← neutral element

Folds the list from right to left

F(a, F(b, F(c, e)))

List(a,b,c).foldLeft(e)(F)

Folds ... from the left

F(F(F(F(e,a),b),c)

The recursion is cleaner, and the way to go

```
def retry(noTimes: Int)(block: => Future[T]): Future[T] = {
  val ns = (1 to noTimes).toList
  val attempts = ns.map(_ => () => block)
  val failed = Future.failed(new Exception("boom"))
  val result = attempts.foldLeft(failed)
    ((a, block) => a recoverWith { block() })
}
```

```
// or
val result = attempts.foldRight(() => failed)
  ((block, a) => () => { block fallbackTo { a() } })
result()
```