# A Simple FRP Implementation

We will now develop a simple implementation of Signals and Vars, which together make up the basis of our approach to functional reactive programming.

## Implementation Idea

- Each signal maintains:
  - Its current value
  - The current expression that defines the signal value
  - A set of observers: the other signals that depend on its value
- Then, if the signal changes, all observers need to be re-evaluated

## Dependency Maintenance

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 a sig() means adding caller to the observers of sig
- When signal sig's value changes, all previously observing signals are re-evaluated and the set sig.observers is cleared
- Re-evaluation will re-enter a calling signal caller in sig.observers, as long as caller's value still depends on sig

## Who's Calling?

- How do we find out on whose behalf a signal expression is evaluated?
- One simple way to do this is to maintain a global data 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

## Stackable Variables

- Here's a class for stackable variables:

```
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
  }
}
```

- You access it like this:

```
val caller = new StackableVar(initialSig)
caller.withValue(otherSig){ … }
… caller.value …
```

## Set Up in Object Signal

- We also evaluate signal expressions at the top-level when there is no other signal that's defined or updated
- We use the "sentinel" object NoSignal as the caller for these expressions
- This how the API of the Signal will look like:

```
object NoSignal extends Signal[Nothing](???) { … }
// there is no need for an implementation for NoSignal
object Signal {
  val caller = new StackableVariable[Signal[_]](NoSignal)
  def apply[T](expr: => T) = new Signal(expr)
}
```

## The Signal Class

```
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 = {
  // gets called during the initialization of a signal and whenever somebody
  // assigns a new value to the signal
    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
  }
}
```

**Exercise:** *The Signal class still lacks an essential part. Which is it?* - ***Reevaluating callers***

## Reevaluating Callers

- A signal's current value can change when:
  - Somebody calls an update operation on a Var
  - The value of a dependent signal changes

- Propagating changes 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())
    }
}
```

## Handling NoSignal

- computeValue needs to be disabled for NoSignal because we cannot evaluate an expression of type Nothing:

```
object NoSignal extends Signal[Nothing](???) {
```

- `override def computeValue() = ()`

```
}
```

## Handling Vars

- Recall that Var is a Signal which can be updated by the client program
- In fact, all necessary functionality is already present in class Signal; we just need to expose it:

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

## Discussion

- Our implementation of FRP is quite stunning in its simplicity. But you might argue that it is too simplistic
- In particular, it makes use of the worst kind of state: global state:

```
object Signal {
    val caller = new StackableVariable[Signal[_]](NoSignal)
}
```

- One particular problem is: What happens if we try to evaluate several signal expressions in parallel? – *Multiple threads could access caller at the same time and could also update caller and stackable variable, so without protection in terms of synchronization, we would get race conditions and unpredictable results.*

## Thread-Local State

- 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 thread-local state
- Thread-local state means that each thread accesses a separate copy of a variable

- It is supported in Scala through class ***scala.util.DynamicVariable***
- The API of DynamicVariable matches the one of StackableVariable, so we can simply swap it into our Signal implementation:

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

## Another Solution: Implicit Parameters

- Thread-local state comes with a number of disadvantages:
    - Its imperative nature often produces hidden dependencies which are hard to manage
    - Its implementation on the JDK involves a global hash table lookup, which can be a performance problem
    - It does not play well in situations where threads are multiplexed between several tasks
- So to summarize, thread-local state is an improvement of unprotected global state, but it has its own set of problems. It's fragile and it plays well only with some approaches to concurrency. And it has still the problem that if that it is fundamentally a state that is shared by a large part of the application
- A cleaner solution involves implicit parameters:
    - Instead of maintaining a thread-local variable, pass its current value into a signal expression as an implicit parameter
    - This is purely functional
    - In current Scala it requires more boilerplate than the thread-local solution, but future versions of Scala might solve that problem

## Summary

- We have given a quick tour through functional reactive programming, with some usage examples and an implementation
- This is just a taster, there's much more to be discovered in the field
- In particular, we only covered one particular style of FRP: Discrete signals changed by events. Some variants of FRP also treat continuous signals
- Values in three systems are often computed by sampling instead of event propagation