

# Ozma: Extending Scala with Oz Concurrency



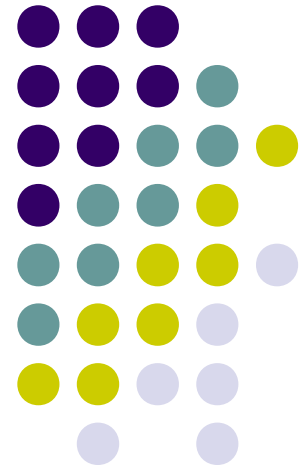
**Sébastien Doeraene**  
**@sjrdoeraene**  
**Peter Van Roy**

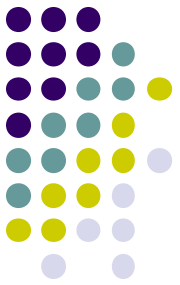
Strange Loop 2012, St. Louis

Sep. 25, 2012

PLDC Research Group  
([pldc.info.ucl.ac.be](http://pldc.info.ucl.ac.be))

Université Catholique de Louvain  
B-1348 Louvain-la-Neuve, Belgium



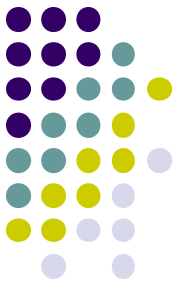


# Introductory example

- Let us compute two difficult numbers, add them, and display the result
- Sequential (aka, obsolete) version:

```
val x = computeToughNumber1()  
val y = computeToughNumber2()  
val z = x+y  
println(z)
```
- Let us now compute x and y concurrently, in the hope that a modern computer (or network of computers) can parallelize the computations
  - A bit of history ...

# Java monitors



```
private final TestMonitors self = this;

private boolean xDone = false;
private int xValue = 0;
private boolean yDone = false;
private int yValue = 0;
private boolean zDone = false;
private int zValue = 0;

private void run()
    throws InterruptedException {
    new ComputeX().start();
    new ComputeY().start();
    new ComputeZ().start();

    final int z;

    synchronized (this) {
        while (!zDone)
            wait();
        z = zValue;
    }

    System.out.println(z);
}

private class ComputeX extends Thread {
    public void run() {
        final int x = 1;
        synchronized (self) {
            xValue = x;
            xDone = true;
            self.notifyAll();
        }
    }
}

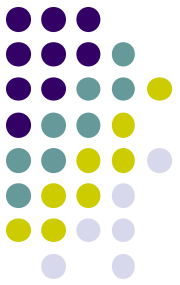
private class ComputeY extends Thread {
    public void run() {
        final int y = 2;
        synchronized (self) {
            yValue = y;
            yDone = true;
            self.notifyAll();
        }
    }
}

private class ComputeZ extends Thread {
    public void run() {
        try {
            final int x, y;

            synchronized (self) {
                while (!xDone || !yDone)
                    wait();
                x = xValue;
                y = yValue;
            }

            final int z = x + y;

            synchronized (self) {
                zValue = z;
                zDone = true;
                self.notifyAll();
            }
        } catch (InterruptedException error) {
            // arg... what do I do now?
        }
    }
}
```



# Java executors and futures

```
private void run() throws Exception {
    ExecutorService executor =
        Executors.newFixedThreadPool(4);

    Future<Integer> xFuture =
        executor.submit(new ComputeX());
    Future<Integer> yFuture =
        executor.submit(new ComputeY());
    Future<Integer> zFuture =
        executor.submit(new ComputeZ(
            xFuture, yFuture));

    System.out.println(zFuture.get());
}
```

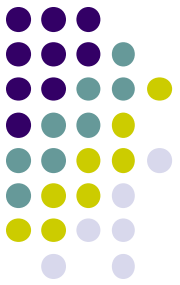
```
private class ComputeX
    implements Callable<Integer> {
    public Integer call() {
        return 1;
    }
}
```

```
private class ComputeY
    implements Callable<Integer> {
    public Integer call() {
        return 2;
    }
}
```

```
private class ComputeZ
    implements Callable<Integer> {
    private final Future<Integer> xFuture;
    private final Future<Integer> yFuture;

    public ComputeZ(Future<Integer> xFuture,
        Future<Integer> yFuture) {
        this.xFuture = xFuture;
        this.yFuture = yFuture;
    }

    public Integer call() throws Exception {
        return xFuture.get() + yFuture.get();
    }
}
```

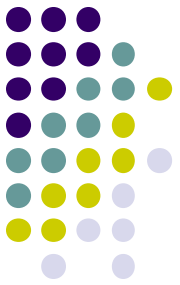


# Scala runners and futures

```
val x = future(1)
val y = future(2)
val z = future(x() + y())

println(z())
```

- Much better!
- Two remaining issues
  - Need to write `x()` instead of just `x` to read the value
  - Blocking: this example uses 4 OS threads on its own, but they are blocking most of the time



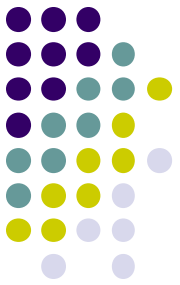
# Scala futures of SIP-14

```
val xFut = future(1)
val yFut = future(2)
```

```
val zFut = for {
  x <- xFut
  y <- yFut
} yield {
  x + y
}
```

```
zFut onSuccess {
  println(_)
}
```

- Designed to solve the blocking issue
- However, the syntax gets trickier again
- Forces the programmer to think asynchronously
- Challenge: can we do better?

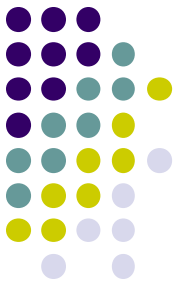


# Ozma

```
val x = future(1)
val y = future(2)
val z = future(x+y)
```

```
println(z)
```

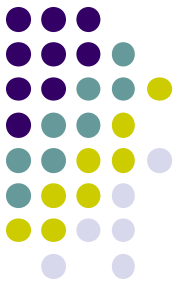
- Easy as can be
- No need for `x():` the type of `x` is `Int`, not `Future[Int]`
  - The future behavior is inside the **language** (dataflow)
- Ozma threads are **lightweight**, i.e., they are not OS threads
  - The blocking issue does not appear
  - What appears to be blocking is actually posting to the dataflow variable a continuation with the remaining of the thread's job



# Overview of the talk

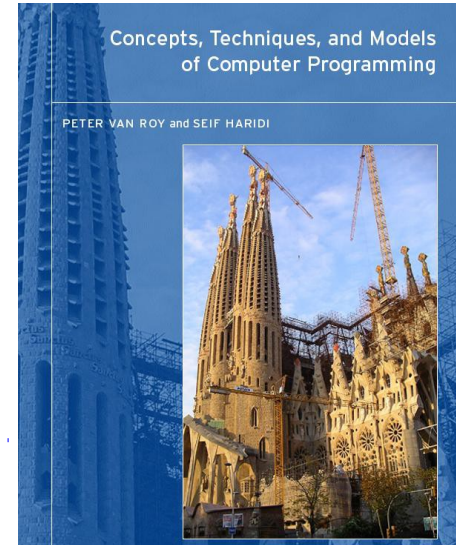
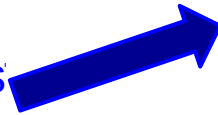
- Scala + Oz  $\Rightarrow$  Ozma
- Declarative dataflow
  - Lightweight threads and the wonders of single assignment `val`
  - Three powerful principles
- Message passing and nondeterminism
  - This is also very important, so let's add it cleanly
- Implementation on the JVM
  - Issues, solutions and work-arounds
- Conclusion
  - The future of Ozma, distribution, and fault tolerance



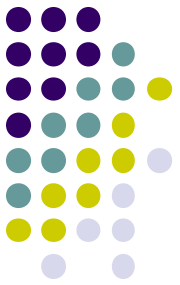


# Scala + Oz $\Rightarrow$ Ozma

- Oz is a multiparadigm language that has been used for language experiments by a bunch of smart but eccentric language researchers since the early 1990s (see [www.mozart-oz.org](http://www.mozart-oz.org))
    - Constraint programming, network-transparent distributed programming, declarative/procedural GUI programming, concurrent programming
    - Textbook “Concepts, Techniques, and Models of Computer Programming”, MIT Press, 2004
  - Oz supports concurrent programming based on a declarative dataflow core with lightweight threads
- $\Rightarrow$  Ozma extends Scala with a new concurrency model based on the Oz dataflow ideas



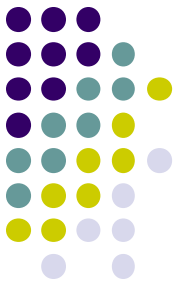
One third of the book is about concurrency



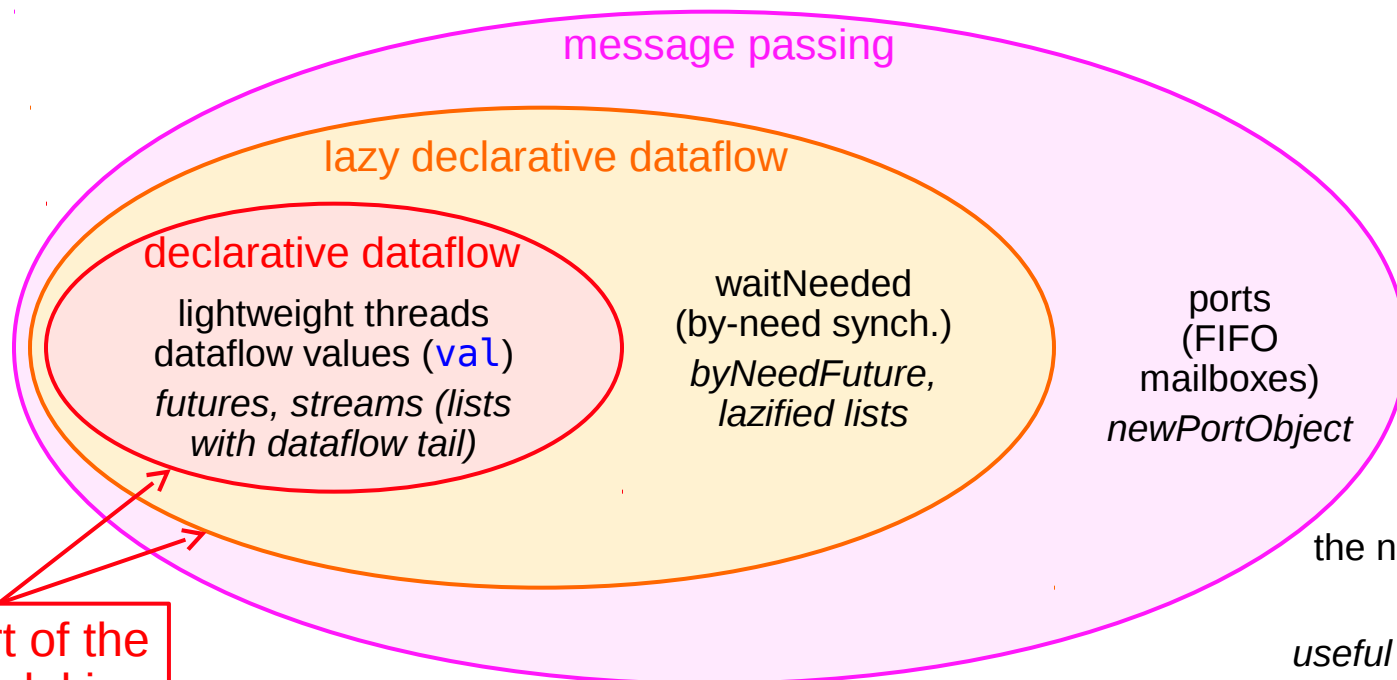
# Ozma implementation

- Ozma's implementation combines a modified Scala compiler and a modified Oz compiler, and targets the Oz VM (Mozart). It was first released in June 2011.
  - The Oz VM has efficient support for lightweight threads, dataflow synchronization, by-need synchronization, and failed values
- Full source and binaries (with open-source license) available at:  
<https://github.com/sjrd/ozma>
- Full documentation available at:  
<http://www.info.ucl.ac.be/~pvr/MemoireSebastienDoeraene.pdf>
- Download the compiled binaries and try it out!
  - Or compile it yourself with Scala  $\geq 2.9.0$ , Mozart  $\geq 1.4.0$ , and Ant  $\geq 1.6$
  - It runs under Linux, Mac OS X, and maybe Windows
- All the Ozma examples in this talk are running code

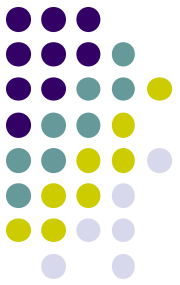
# Ozma extends Scala with a new concurrency model



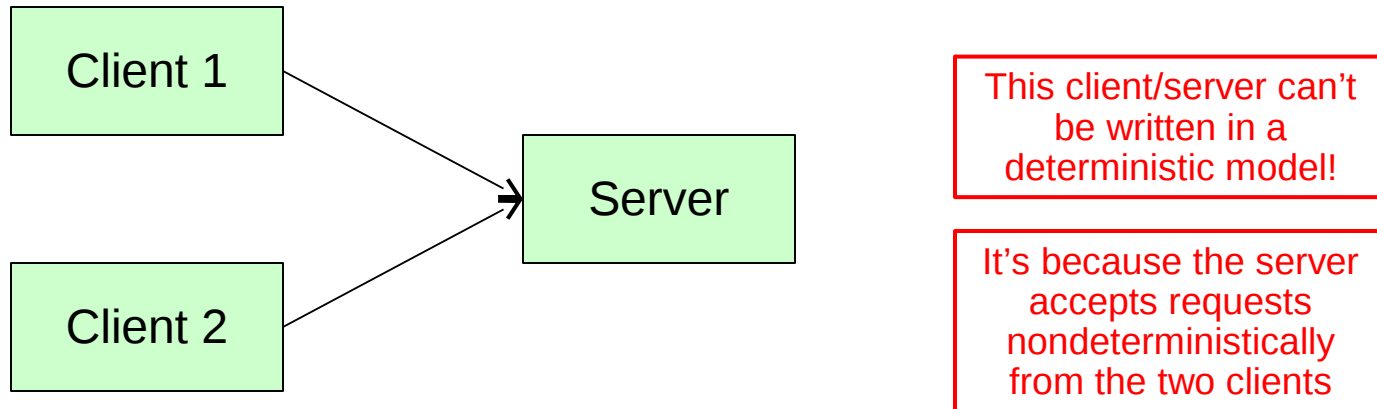
- The heart of the model is declarative dataflow
  - Further extended with laziness (still declarative) and ports (for nondeterminism)
  - This allows adding nondeterminism exactly where needed and no more



The heart of the new model is ***deterministic***

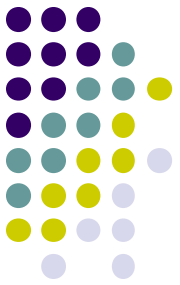


# Why deterministic concurrency?



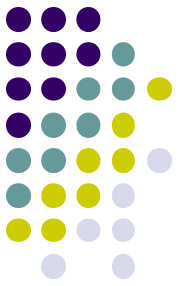
- Determinism has strong limitations!
  - Any concurrent execution always gives the same results
  - Even a simple **client/server can't be written**
- But determinism has big advantages too
  - **Race conditions are impossible** by design
  - With determinism as default, we can **reduce the need for nondeterminism** (in the client/server: it's needed only at the point where the server accepts requests)
  - **Any functional program can be made concurrent** without changing the result

# Deterministic concurrency: the right default?



- Parallel programming has finally arrived
  - **Multicore processors**: dual and quad today, a dozen tomorrow, a hundred in a decade, most apps will do it
  - **Distributed computing**: data-intensive with tens of nodes today (NoSQL, MapReduce), hundreds and thousands tomorrow, most apps will do it
- Something fundamental will have to change
  - Sequential programming can't be the default (it's a centralized bottleneck)
  - Libraries can only hide so much (interface complexity, distribution structure)
- Concurrency will have to get a lot easier
  - Deterministic concurrency is functional programming!
  - It can be extended cleanly to distributed computing
    - Open network transparency (implemented in Oz since 1999)
    - Modular fault tolerance (implemented in Oz since 2007)
    - Large-scale distribution (on the way...)

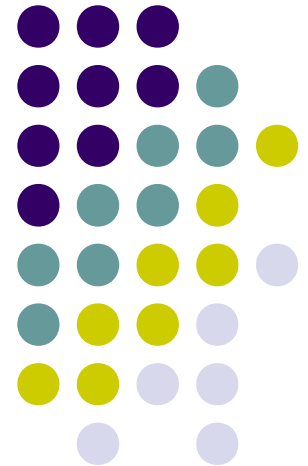
# Such an old idea, why isn't it used already?



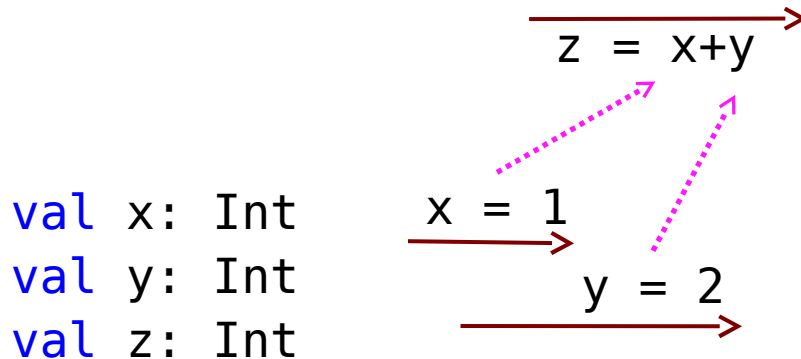
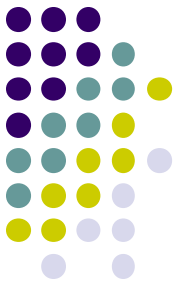
- **Deterministic concurrency** has a long history that starts in 1974
  - Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pp. 471-475, 1974. *Deterministic concurrency*.
  - Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pp. 993-998, 1977. *Lazy deterministic concurrency*.
- Why was it forgotten for so long?
  - Message passing and monitors arrived at about the same time:
    - Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3<sup>rd</sup> International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235-245, Aug. 1973.
    - Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549-557, Oct. 1974.
  - **Actors and monitors handle nondeterminism, so they are better. Right?**
- **Dataflow computing** also has a long history that starts in 1974
  - Jack B. Dennis. First version of a data flow procedure language. *Springer Lecture Notes in Computer Science*, vol. 19, pp. 362-376, 1974.
  - **Dataflow remained a fringe subject since it was always focused on parallel programming,** which only became mainstream with the arrival of multicore processors in mainstream computing (e.g., IBM POWER4, the first dual-core processor, in 2001).

# Declarative Dataflow

---



# Declarative dataflow



→ Thread execution  
(executes from left to right)

.....→ Dataflow synchronization

```
thread { x = 1 }  
thread { y = 2 }  
thread { z = x+y }
```

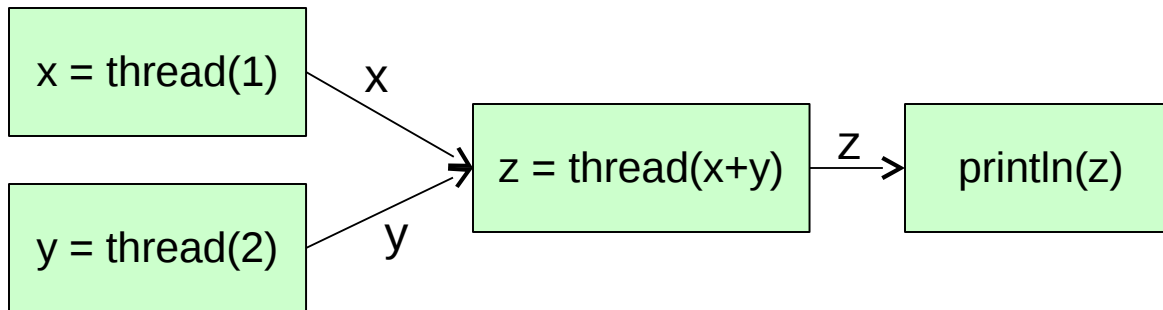
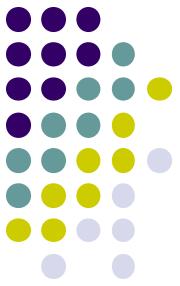
```
println(z)
```

- All **val** values can do dataflow
- They are single assignment
- The addition operation *waits* until both x and y are bound
- This does both synchronization and communication

- Programs with declarative dataflow are **always deterministic**
- This program will always print 3, independent of the scheduler



# Using the thread statement as an expression



Each green box is  
a concurrent agent

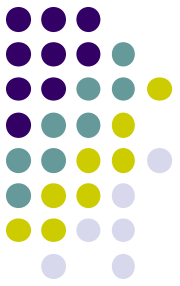
Each arrow is a  
shared dataflow value

```
val x = thread(1)
val y = thread(2)
val z = thread(x+y)

println(z)
```

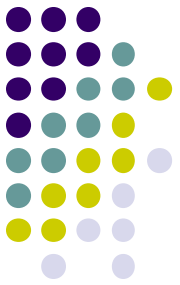
- Exactly the same behavior as the previous example
- Using the thread statement in this way can often simplify the syntax of concurrent programs

# Handling exceptions in asynchronous computations



```
try {  
  val list: List[Int] = Nil  
  val x = thread(list.head) // list is empty!  
  println(x)  
} catch {  
  case _: java.util.NoSuchElementException =>  
    println("The list was empty")  
}
```

- What happens if the asynchronous computation (in `thread`) throws an exception?
- The only reasonable possibility is to raise the exception where `x` is needed
  - Well-known behavior of **futures**



# Futures and failed values

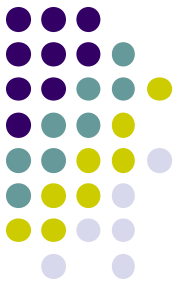
```
def future[A](value: => A): A = { // value is by-name
  thread {
    try {
      value // value is evaluated here
    } catch {
      case NonFatal(throwable) =>
        makeFailedValue(throwable)
    }
  }
}
```

- If the evaluation of `value` throws an exception, the exception is wrapped in a **failed value** using the Ozma primitive `makeFailedValue`
- Waiting for a failed value throws the wrapped exception
- A failed value has type `Nothing`, the bottom type of Scala
- Now we can write:

```
val x = future(list.head)
```

and the exception will be properly propagated to the current thread

# Declarative dataflow extensions to Scala



- **Lightweight threads**: hundreds of thousands of threads can be active simultaneously (like Erlang, by the way)

```
thread { println("New lightweight thread") }
```

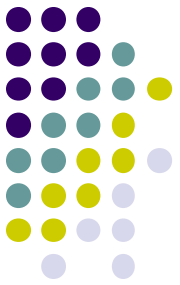
- **Dataflow values**: every **val** can be a single-assignment variable. Operations that need the value will wait until it is available.

```
val x = thread(1) // binds x in its own thread
println(x+10)      // the addition waits for x
```

- **By-need (lazy) execution**: wait until value is *needed*

```
val x: Int
thread { waitNeeded(x); x = factorial(69) }
println(x) // need to print causes calculation of x
```

# Streams: lists as dataflow communication channels



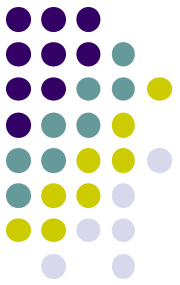
```
val x: List[Int]
val ints = 1 :: 2 :: 3 :: 4 :: x // unbound tail

thread { ints foreach println } // a printing agent

val y: List[Int]
x = 5 :: 6 :: 7 :: y // the agent will print these
```

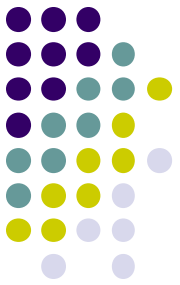
- A stream is a **list with an unbound dataflow tail**
  - It can be extended indefinitely or terminated with `Nil`
- Any list function can read a stream (it's exactly like reading a list)
  - It will automatically wait when it finds an unbound tail
    - Like the `foreach` operation in this example
  - If put inside a thread, the list function becomes a **concurrent agent**

# The magic of declarative dataflow

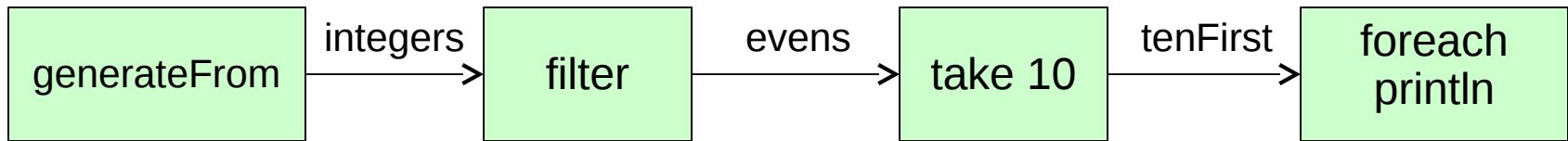


```
object Test {  
  def main(args: Array[String]) {  
    val range = gen(1, 10) // sequential version  
    val result = range map (x => x*x)  
    result foreach println  
  
    val range2 = thread(gen(1, 10)) // concurrent version  
    val result2 = thread(range map (x => x*x))  
    result2 foreach println  
  }  
  def gen(from: Int, to: Int): List[Int] = {  
    sleep(1000)  
    if (from > to) Nil  
    else from :: gen(from+1, to) // tail-recursive in Ozma  
  }  
}
```

- Both versions print the same final result 1, 4, 9, 16, ..., 100
  - So what's the difference? What does concurrency buy you?
- The sequential version: nothing is output for 10 seconds, and then the whole list
- The concurrent version: a new result is output every second
- Declarative dataflow turns batch programs into incremental programs



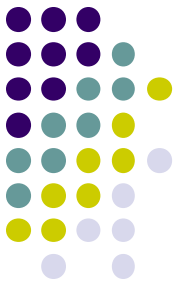
# Pipelines using streams



```
def generateFrom(n: Int): List[Int] =  
  n :: generateFrom(n+1)
```

```
val integers = thread(generateFrom(0))  
val evens = thread(integers filter (_ % 2 == 0))  
val tenFirst = thread(evens take 10)  
tenFirst foreach println
```

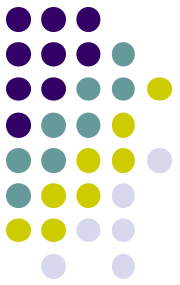
- A list function put in a thread becomes a concurrent agent
- List functions must be tail-recursive for this to work
  - This is automatically true in Ozma (ensured by compiler transformation)



# Three powerful principles


- Any functional program can be made concurrent without changing the result by adding calls to thread
  - Threads can be added anywhere in the program
  - Turns batch into incremental (removes roadblocks)
- Any list function can become a concurrent agent by executing it in a thread
  - Because list functions in Ozma are tail-recursive, the agent has no memory leak (stack size and heap size are constant)
- Any computation, functional or not, can be made lazy by adding calls to `waitNeeded`
  - Syntactic sugar is provided with `byNeedFuture` and `.lazified`





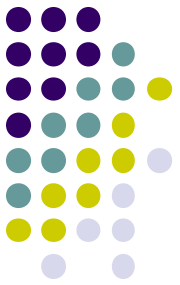
# From map to concurrent map

```
def map[A, B](list: List[A], f: A => B): List[B] = {  
  if (list.isEmpty) Nil  
  else f(list.head) :: map(list.tail, f)  
}
```



```
def concMap[A, B](list: List[A], f: A => B): List[B] = {  
  if (list.isEmpty) Nil  
  else thread(f(list.head)) :: concMap(list.tail, f)  
}
```

- In concMap, all evaluations of f execute concurrently
- It is even possible to call concMap when f is not known (unbound). This will create a list containing unbound values, like futures: they will be evaluated as soon as f is known (bound to a function).



# Map as a concurrent agent

```
def gen(from: Int): List[Int] = from :: gen(from+1)
```

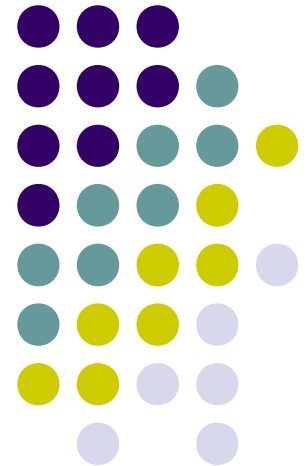
```
def displayEvenSquares() {  
  val integers = thread(gen(0))  
  val evens = thread(integers filter (_ % 2 == 0))  
  val evenSquares = thread(evens map (x => x*x))  
  evenSquares foreach println  
}
```

Concurrent agent

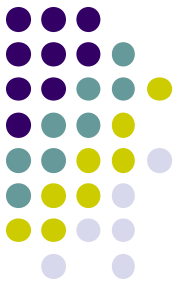
- Wrapping the calls to `gen`, `filter`, and `map` within threads turns them into concurrent agents
  - Note that `foreach` is also an agent, living in the main thread
- As new elements are added to the input stream, new computed elements will appear on the output stream

# Message Passing and Nondeterminism

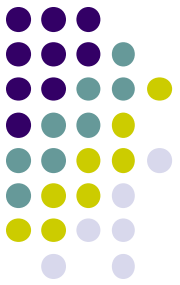
---



# Managing nondeterminism with ports



- So far, all our programs have been deterministic
  - Determinism is a good default, but for real programs we need nondeterminism too!
- Let's add nondeterminism in a nice way
  - One way is to allow multiple producers (or clients) to add messages in a single stream (read by an agent, or server)
- A **port** is comparable to an unbounded FIFO mailbox
  - Any thread can send a value to a port
  - There is no receive operation; all messages appear in an associated stream
  - The senders and the receivers of a port can themselves be deterministic computations; the only nondeterminism is the order in which sent values appear on the port's stream



# Introducing ports

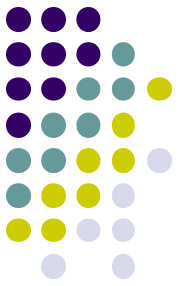
```
val (s, p) = newPort[Int] // Create port p with stream s
```

```
thread { p.send(1) }  
thread { p.send(2) }  
thread { p.send(3) }
```

```
thread { s foreach println } // Print elements of the  
                             // port's stream one by one
```

- The values 1, 2, and 3 will be displayed in some order (nondeterminism)
  - The actual order depends on the thread scheduler
- No memory leak: garbage collection will remove the parts of the stream already read

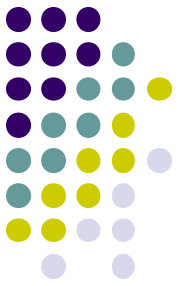
# Merging two streams that are fed concurrently (broken)



```
def mergeStreams[A](s1: List[A], s2: List[A]): List[A] = {  
  (s1, s2) match {  
    case (h1 :: t1, h2 :: t2) =>  
      h1 :: h2 :: mergeStreams(t1, t2)  
  }  
}
```

- Does not work if the two streams do not grow exactly at the same pace
- Fundamental issue: we cannot know a priori from which stream the following value will come (nondeterminism)
- A port solves exactly this problem

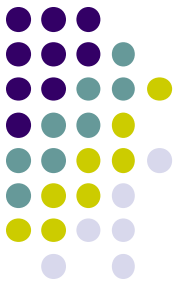
# Merging two streams that are fed concurrently (correct)



```
def mergeStreams[A](s1: List[A], s2: List[A]): List[A] = {  
  val (result, p) = newPort[A]  
  thread { s1 foreach p.send }  
  thread { s2 foreach p.send }  
  result  
}
```

- Two declarative agents read the input streams, and forward messages into the port
- The port accepts elements from both inputs in a nondeterministic order (dependent on time and scheduler)

# Building nondeterministic agents with ports



```
def newPortObject[A, B](init: B)(  
  handler: (B, A) => B) = {  
  val (s, p) = Port.newPort[A]  
  thread { s.foldLeft(init)(handler) }  
  p  
}
```

Initial state

State updater

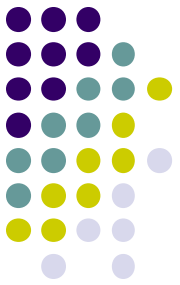
- A port object is an actor. It reads messages sequentially from the stream and uses the messages to update its internal state.
- The `foldLeft` operation updates the internal state as messages are received (note:  $s_i$  is a received message):

$(\dots(((\text{init handler } s_0) \text{ handler } s_1) \text{ handler } s_2) \dots )$

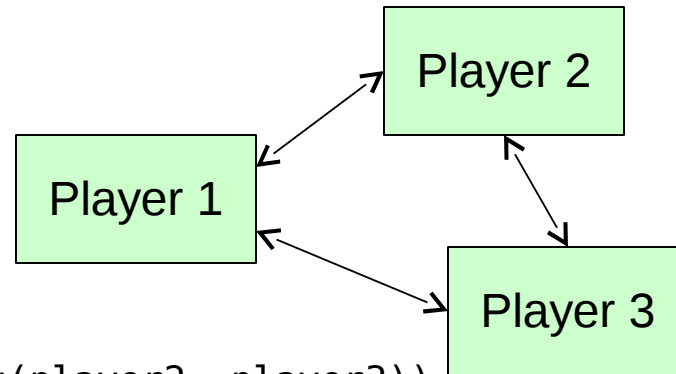
- The current value of the accumulator of `foldLeft` is the agent's internal state
- Neat trick: `foldLeft` is a **function used as a concurrency pattern**



# Agents playing ball

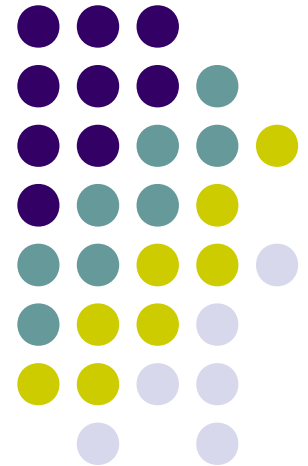


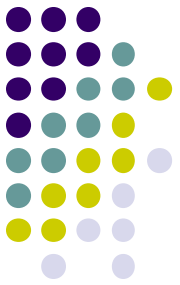
```
object BallGame {
  type Ball = Unit
  val ball: Ball = ()
  type Player = Port[Ball]
  def main(args: Array[String]) {
    val player1: Player
    val player2: Player
    val player3: Player
    player1 = makePlayer("Player 1", Seq(player2, player3))
    player2 = makePlayer("Player 2", Seq(player3, player1))
    player3 = makePlayer("Player 3", Seq(player1, player2))
    player1.send(ball)
    while (true) sleep(1000)
  }
  def makePlayer(id: Any,
    others: Seq[Player]): Player = {
    Port.newPortObject(0) { (st: Int, b: Ball) =>
      println("%s received the ball %d times"
        format (id, st+1))
      Random.rand(others).send(b)
      st+1
    }
  }
}
```



- Each player receives the ball and sends it to a randomly chosen other player
- Each player counts the number of balls received
- The port allows a player to receive from either of the others (nondeterminism)

# Ozma on the JVM





# Core requirements

- Every `val` must be dataflow-enabled
  - Single-assignment
  - Implicit synchronization
  - Failed values
- Threads should be lightweight
  - Programming techniques of Ozma encourage to spawn many threads
  - Blocking should be avoided: waiting for an unbound value should post a continuation to the value's suspension list
  - As far as we know, there is no way to emulate lightweight threads with the current JVM
  - Ideas welcome!

# Implementing dataflow



```
trait Dataflow[@specialized +A] {  
  def ask: A  
}
```

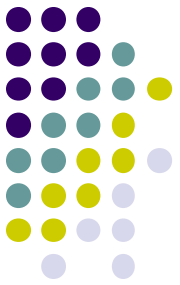
```
class DataflowVar[@specialized A] extends Dataflow[A] {  
  private var value: A = _  
  private var bound = false
```

```
  def this(v: A) {  
    this()  
    value = v  
    bound = true  
  }
```

- A Dataflow[T] looks like a blocking Future[T]
- A DataflowVar[T] looks like a Promise[T] (plus the corresponding Future[T])

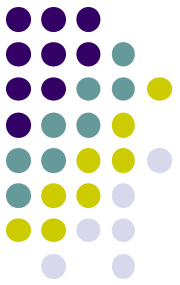
```
  def tell(v: A): Unit = ???  
  def ask: A = ???  
}
```

# Implementing dataflow



```
def tell(v: A) {  
  synchronized {  
    if (!bound) {  
      value = v  
      bound = true  
      notifyAll()  
    } else if (value == v) {  
      // telling twice the same thing is OK  
    } else {  
      // failure (not declarative!)  
      throw new FailureError(value, v)  
    }  
  }  
}
```

# Implementing dataflow



```
def ask: A = {  
  synchronized {  
    while (!bound)  
      wait()  
    value  
  }  
}
```

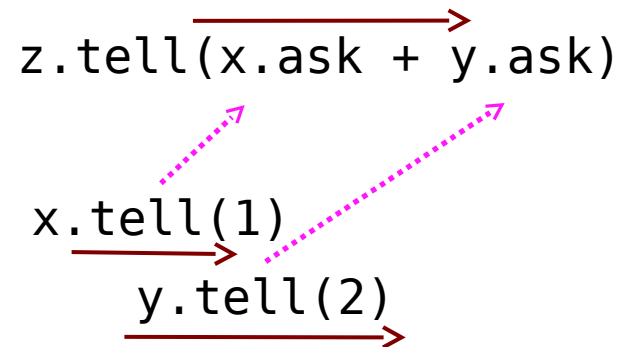
```
def thread(body: => Unit) {  
  new Thread() {  
    override def run() = body  
  }.start()  
}
```

# Using DataflowVar[A]

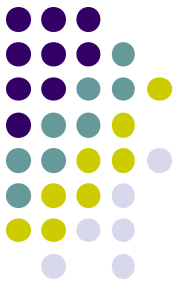
```
val x = new DataflowVar[Int]
val y = new DataflowVar[Int]
val z = new DataflowVar[Int]

thread { x.tell(1) }
thread { y.tell(2) }
thread { z.tell(x.ask + y.ask) }

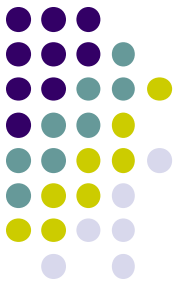
println(z.ask)
```



- We lose transparency, of course
- Can be improved with implicit conversions of A to DataflowVar[A] and from Dataflow[A], but it is still limited



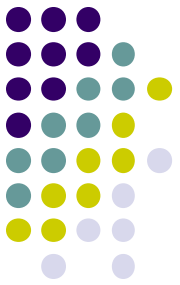
# Implementing thread



```
def thread[@specialized A](  
    body: => Dataflow[A]): Dataflow[A] = {  
    val result = new DataflowVar[A]  
    new Thread() {  
        override def run() = result.tell(body.ask)  
    }.start()  
    result  
}
```



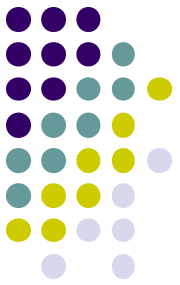
# Implementing Port



```
class Port[-A] private (  
  stream: DataflowVar[DataflowList[A]] {  
    private var tail: DataflowVar[DataflowList[A  
      @uncheckedVariance]] = stream  
  
  def send(element: A) {  
    val newTail = new DataflowVar[DataflowList[A]]  
    val cons = element :: newTail  
    synchronized {  
      tail.tell(cons)  
      tail = newTail  
    }  
  }  
}
```

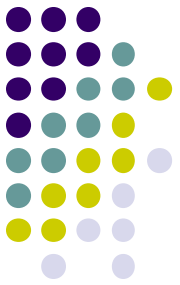
- A `DataflowList[A]` is akin to a `List[A]`, but its tail is itself a `Dataflow[DataflowList[A]]`

# Implementing Port



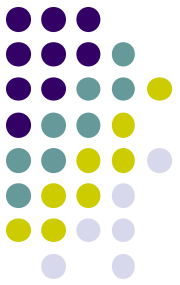
```
object Port {  
  def newPort[A]: (Stream[A], Port[A]) = {  
    val stream = new DataflowVar[DataflowList[A]]  
    val port = new Port[A](stream)  
    (stream, port)  
  }  
}
```

```
type Stream[+A] = Dataflow[DataflowList[A]]
```



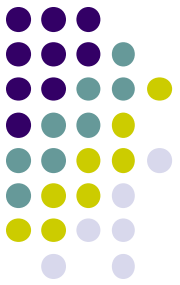
# Core requirements revisited

- Every `val` must be dataflow-enabled
  - Every variable of type `T` should be a `Dataflow[T]`
  - Every single-assignment `val` of type `T` should be a `DataflowVar[T]`
  - Consequence: no need for `DataflowList[A]`, since the tail of `List[A]` is implicitly a `Dataflow[List[A]]`.
- Can be achieved by compiler transformations!
  - Modify scalac to add these transformations



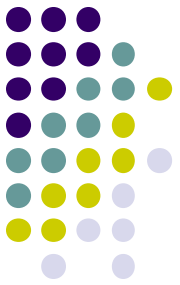
# scalac in a nutshell

- The Scala compiler consists of several transformation phases
- Front-end phases
  - The parser builds an untyped AST from the source code
  - `namer`, `packageobjects` and `typer` yield a typed AST
- Simplifying phases
  - Various phases successively simplify the typed AST until only Java-like classes and constructs remain
  - One particular phase is worth mentioning: `erasure`, which eliminates all the generic types
- Back-end phases
  - `icode` turns the simplified typed AST into a portable stack-based bytecode called the I-code
  - Several optimization phases
  - `genjvm` turns the I-code into JVM bytecode and `.class` files



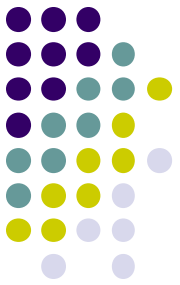
# Naive transformation

- Add a phase dataflow in the compiler between `tailcalls` and `specialize` (the latter being itself just before erasure).
- Do not touch subclasses of `AnyVal`, nor `Dataflow[A]` and `DataflowVar[A]` themselves.
- Retype all Scala-declared variables, fields, parameters and return values from their type `T` to `Dataflow[T]`.
- Retype single-assignment `val`'s of type `T` to `DataflowVar[T]`, and initialize them with a `new DataflowVar[T]`.
- Turn assignments to single-assignment `val`'s into calls to `tell()`.
- Prefix all method calls by `.ask`. Also add `.ask` in `if`'s and `while`'s.
- When calling a native method (e.g., `Int.+`), add `.ask` to all parameters, and wrap the result into a `DataflowVar`.
- And let subsequent phases of the compiler deal with all this.



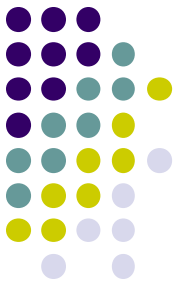
# Erasure and its endless issues

- Basic fact: after erasure, all entities of type `Dataflow[T]` will be retyped as `Dataflow`.
- Type tests with `isInstanceOf` and `asInstanceOf` are broken.
- Pattern matching is therefore also broken.
- Overloads with the same number of arguments, but different types of parameters, erase to the same signature and clash:
  - `foo(x: Int) -> foo(x: Dataflow[Int]) -> foo(x: Dataflow)`
  - `foo(x: Bar) -> foo(x: Dataflow[Bar]) -> foo(x: Dataflow)`



# Working after erasure

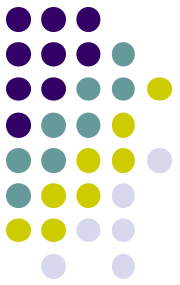
- We do not want to mess with the types before erasure
  - Let us do it after ... actually the later the better (could be just before `icode`)
- Retype all variables, fields, parameters and return values of reference types to `Dataflow`, of type `Int` to `DataflowInt`, etc. (manual specialization)
  - Overload clashes due to the return value are supposed to be only bridge methods, which can be removed in this case
  - We still get overload clashes with parameter types!
- Retype single assignment `val`'s in a similar way to `DataflowVar`
  - No overloading clash here: they are all local variables
- Actually we can forget the `Dataflow` abstraction, and use only `DataflowVar`. Variance checks are behind us anyway.



# Dealing with overloads

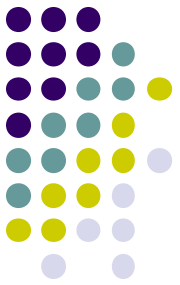
- We still have the following clash:
  - `foo(x: String) -> foo(x: Dataflow)`
  - `foo(x: List) -> foo(x: Dataflow)`
- Three possible workarounds
  - `foo(x: String) -> foo(x: Dataflow, x': String)`
    - Double the number of parameters just for the sake of avoiding overloading clashes
  - `foo(x: String) -> foo(x: DataflowString)`
    - Have a specialized DataflowT class for every class T in the system
  - `foo(x: String) -> foo$java.lang.String(x: Dataflow)`
    - Rename the method to get rid of overloading
    - Probably the best choice





# Interop with Java classes

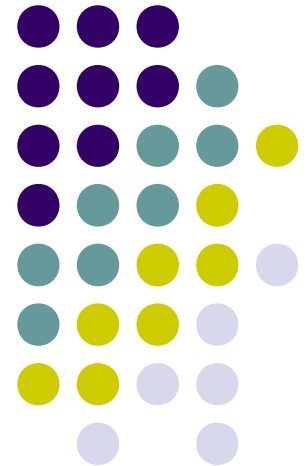
- We have no power over Java-defined classes
  - We cannot change their internals to support dataflow
- When calling a Java method from Ozma code
  - Add `.ask` to parameters and wrap the result in a `DataflowVar`
  - Java classes are considered “native”
- To support calls to Ozma methods from Java code
  - Instead of renaming and retyping methods, duplicate them
  - Keep the original method, and make it call the dataflow-enabled version with the appropriate wrappings and unwrappings in `DataflowVar`'s.
- Interfaces must be duplicated: the original version and the version with dataflow-enabled methods

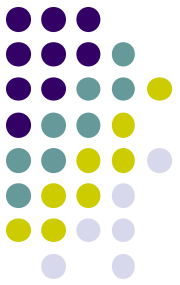


# Ozma on the JVM: is it viable?

- It seems possible to implement Ozma on the JVM
  - Possible, but with an incredible overhead
- Wrapping of all values in DataflowVar's
  - Calls to methods of DataflowVar will likely be inlined by the JVM, but it is a small consolation
- Double the number of methods of every class, to support interoperability with Java classes
  - This includes basic overriding of Java-defined methods
- Threads are not lightweight: we kept the JVM threads
  - The main benefit of Ozma is lost, compared to the existing blocking futures of Scala
- => possible, but probably not practical

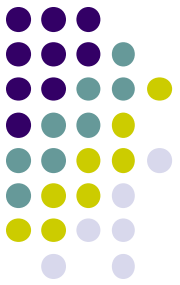
# Conclusion





# Ozma and its model

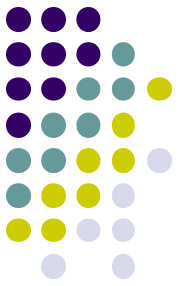
- Ozma makes concurrent programming simpler
  - The heart of a concurrent program is deterministic. Nondeterminism is added just where it's needed.
  - Correctness is easy: the deterministic part is purely functional and the nondeterministic part uses message passing
- The implementation uses the Oz virtual machine (Mozart)
  - It's a complete implementation of Scala on a new VM that's not the JVM nor .NET, so you can see it as a **new implementation of Scala**
  - It's not interoperable with Java, though. The Mozart VM was used because of its support for fine-grain threads, dataflow, and failed values.
  - The upcoming release of Mozart 2 should interoperate a little better with Java.



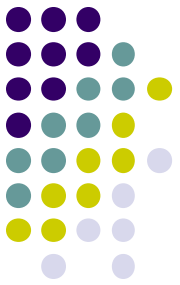
# Ozma on the JVM

- Ozma could be implemented on JVM
  - But with so many restrictions that it would probably not be worth it
- It is still interesting, though
  - Scripting languages like Python are rather slow, and yet used
  - These languages often already have lightweight threads
  - The concepts of Oz and Ozma could be implemented for scripting languages without so much downsides
- Clarke's second law
  - “The only way of discovering the limits of the possible is to venture a little way past them into the impossible.”

# Generalizing dataflow for distribution and fault tolerance



- Language support for distributed programming in Oz
  - **Network transparency**: a program executed over several nodes gives the same result as if it were executed on a single node, provided network delays are ignored and no failure occurs
    - Exact same source code is run independent of distribution structure
  - **Network awareness**: a program can predict and control its physical distribution and network behavior
  - Fully implemented in Oz (Mozart 1.4.0)
- Modular fault tolerance in Oz using fault streams
  - **Exceptions and RMI**: synchronous, not modular, requires changing code at each possible distribution point
  - **Fault streams on language entities**: asynchronous, modular, just add new code with no changes to existing code



# Thank you!