Reactive Programming

J. Heinzelreiter

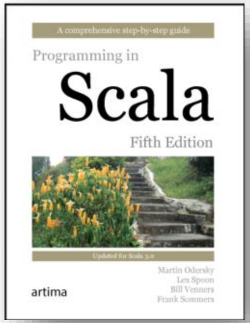
Version 2.3

Content

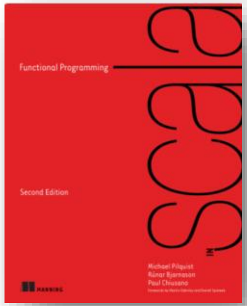
- Part 1: Introduction
  - Motivation for Reactive Programming
  - Traditional Concurrent Programming Techniques
  - Foundations of Functional Programming
- Part 2: Futures and Promises
- Part 3: Actors
  - Foundations of Actor Systems and Akka
  - Designing Actor Systems
  - Actor Supervision
  - Distributed Programming with Actors
- Part 4: Reactive Streams (Akka Streams)

Literature (1)

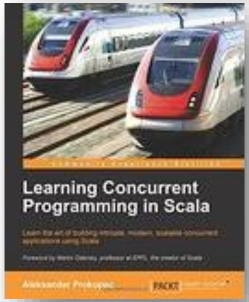
- [Odersky, 2021]: Martin Odersky, Lex Spoon, Bill Venners. **Programming in Scala**. Artima, 5th ed., 2021.



- [Pilquist et al., 2023]: Michael Pilquist, Rúnar Bjarnason, and Paul Chiusano. **Functional Programming in Scala**. Manning, 2nd ed., 2023.

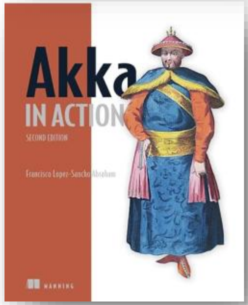


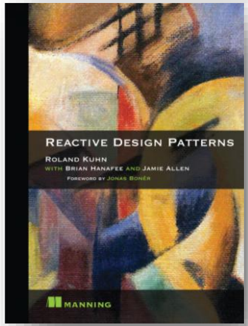
- [Prokopec, 2017]: Aleksandar Prokopec. **Learning Concurrent Programming in Scala**. Packt Publishing, 2nd ed., 2017.



Literature (2)

- [Odersky et al., 2015]: Martin Odersky, Erik Meijer, Roland Khun: [Principles of Reactive Programming](). Online Course, 2015 (see YouTube).
- [Roestenburg et al., 2017]: R. Roestenburg, R. Bakker, R. Williams. **Akka in Action**. Manning, 2017.
- [Lopez-Sancho Abraham, 2023]: F. Lopez-Sancho Abraham. **Akka in Action**. Manning, 2nd ed., 2023.
- [Kuhn, 2017]: R. Kuhn. **Reactive Design Patterns**. Manning, February 2017.



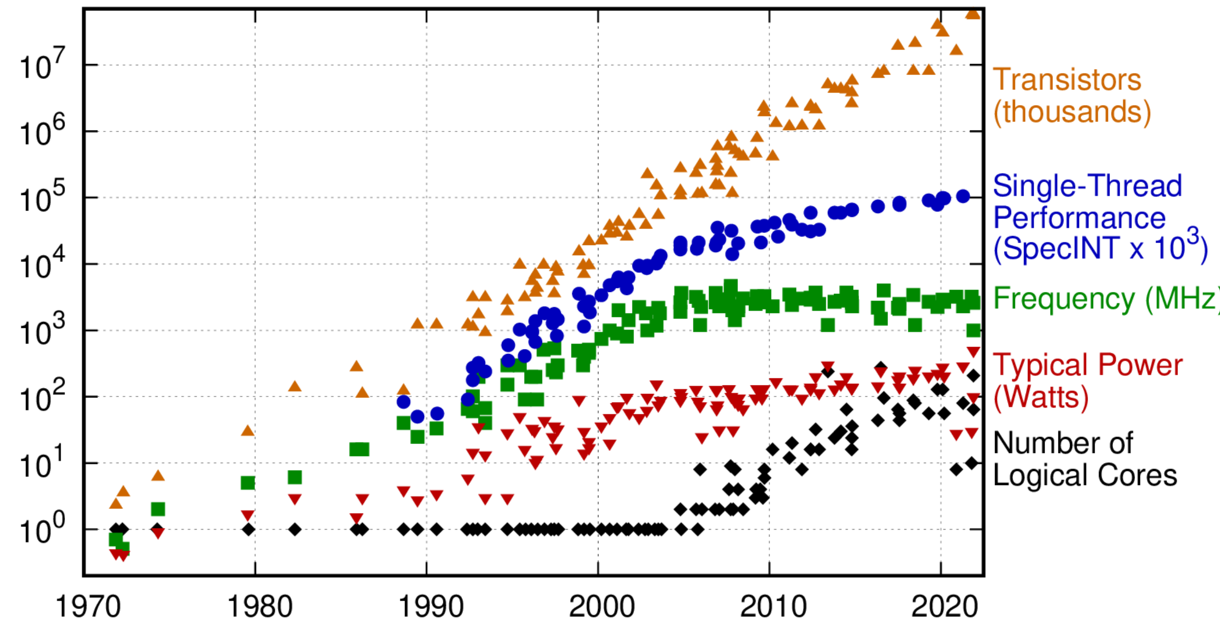


Motivation for
Reactive Programming

[Odersky et al., 2015; What is Reactive Programming?]
[http://www.reactivemanifesto.org]

Moore’s Law

- **Moore's law** is the observation that **the number of transistors** on integrated circuits **doubles** approximately every two years.
- Clock speed doesn’t increase correspondingly anymore.
- Consequences:
  - Shift from single- to multi-core computers.
  - Concurrent programming is becoming more and more important.
  - Modern programming languages are becoming functional.



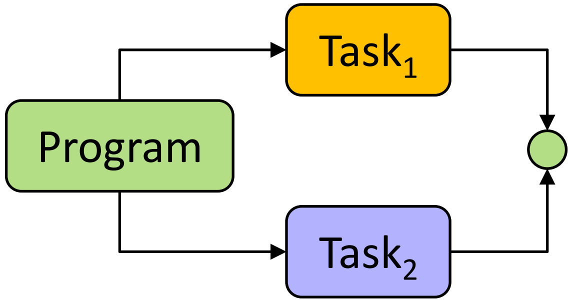
https://github.com/karlrupp/microprocessor-trend-data

“The Free Lunch Is Over”

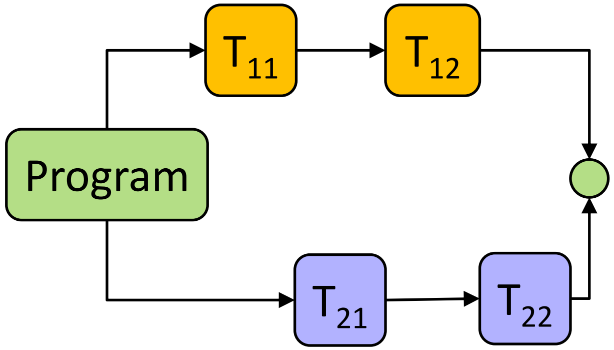
- Article from Herb Sutter (2005): The Free Lunch Is Over.
  A Fundamental Turn Toward Concurrency in Software
  - *Instead of driving clock speeds and straight-line instruction throughput ever higher, they are instead turning en masse to **hyper-threading and multicore architectures**.*
  - ***Applications will increasingly need to be concurrent** if they want to fully exploit continuing exponential CPU throughput gains.*
  - *Efficiency and **performance optimization will get more**, not less, **important**.*
  - *The **vast majority of programmers today don’t grok concurrency**, just as the vast majority of programmers 15 years ago didn’t yet grok objects.*

Concurrent vs. Parallel

- **Parallelism:** Techniques to accelerate programs by performing computations simultaneously.
  - Requires multiple CPUs
  - Computation units must be independent
  - Goal: Increase runtime efficiency

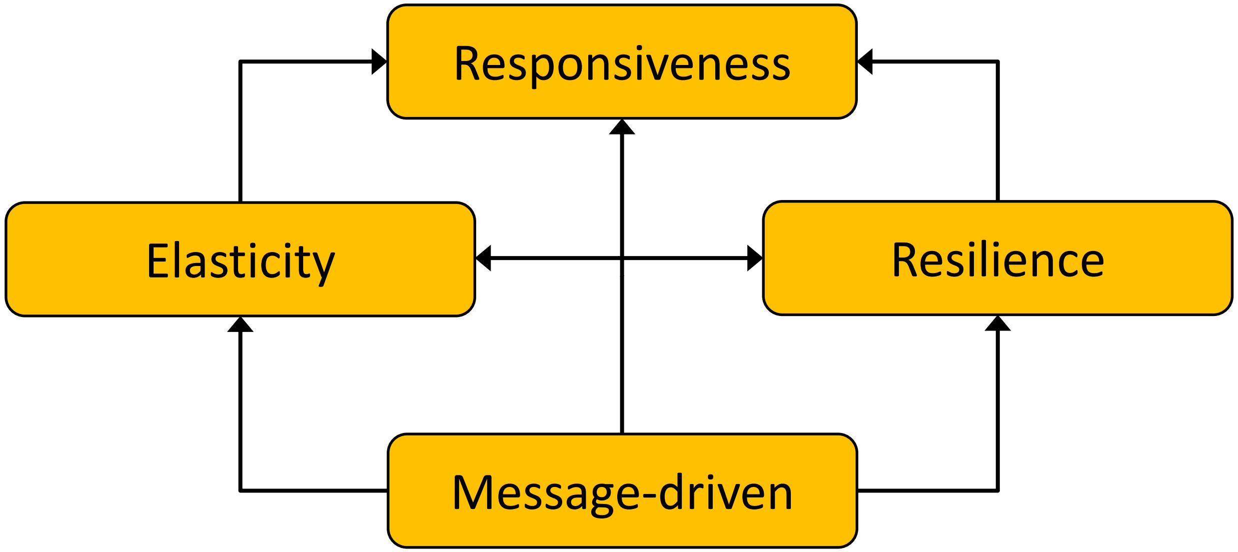


- **Concurrency:** Program makes progress on multiple tasks at the same time.
  - Concurrent programs can be executed on a single CPU, but can benefit from multiple CPUs
  - Goal: Allow efficient interaction with multiple external agents
  - Concurrency includes parallelism



Reactive Systems

- Definition of *reactive*: “Showing a response to a stimulus. Pupils are reactive to light” [Oxford Dictionaries]
- The *Reactive Manifesto*: Characteristics of Reactive Systems



Characteristics of Reactive Systems (1)

- **Message-driven:** react to events
  - Reactive Systems rely on asynchronous message-passing → no blocking.
  - Ensures loose coupling of components → location transparency.
  - Message queues enable load management.
  - Also, failures are communicated via messages.
- **Elasticity:** react to load
  - The system stays responsive under varying workload.
    - *Scale up:* make use of multiple cores, increase memory
    - *Scale out:* make use of multiple server nodes
  - Requirements:
    - Minimize mutable state
    - Location transparency, resilience

Characteristics of Reactive Systems (2)

- **Resilience:** react to failures
  - The system stays responsive in the face of failure.
  - Parts of the system can fail and recover without compromising the system as a whole.
  - Recovery of each component is delegated to another component.
  - Prerequisites:
    - Encapsulation of state
    - Supervisor hierarchies
- **Responsiveness:** react to users
  - System must provide real-time interaction with agents under load and in the presence of failures.
  - Prerequisites: Message-driven architecture, elasticity, resilience.

Traditional
Concurrent Programming Techniques

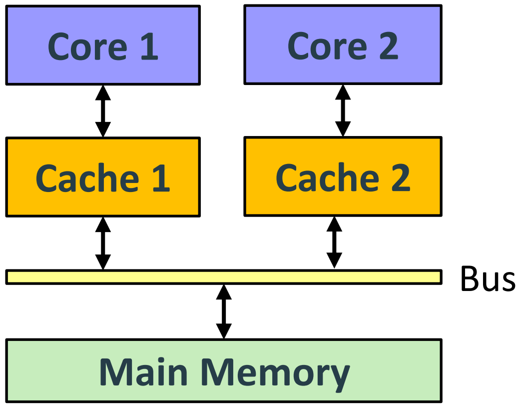
[Prokopec, 2017; p. 27 ff.]

Overview

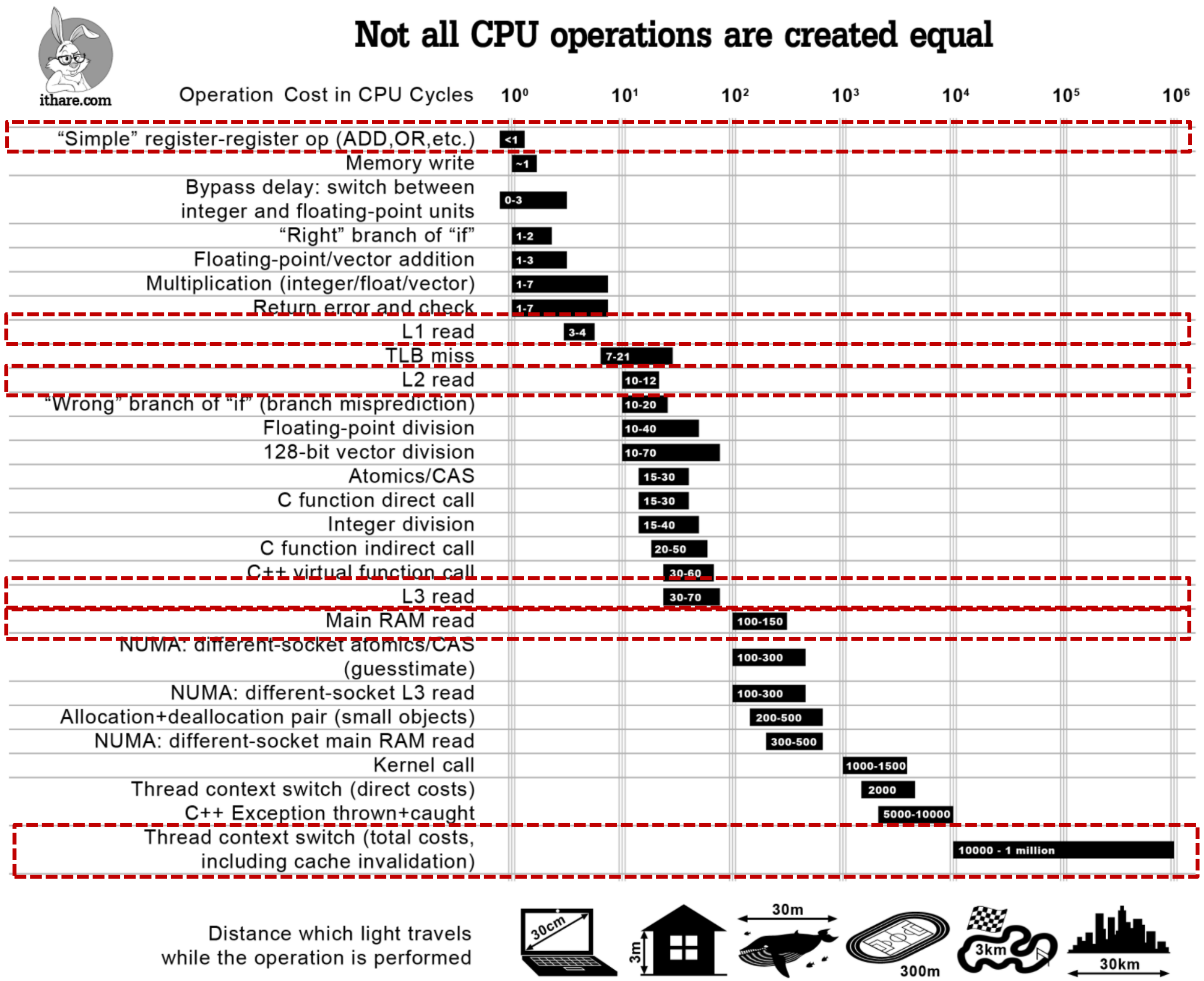
- Traditional low-level constructs for concurrent executions:
  - Processes and
  - Threads
- Means for communication:
  - Shared memory communication → synchronization required
  - Message passing → distributed systems
- Problem: low level and error prone
  - Deadlocks
  - Race conditions
  - Data races
  - Starvation
- Basic knowledge of low-level approaches is essential to understand high-level concepts.

Processes and Threads

- A *process* executes the instructions of a program.
  - Shares CPU and other resources with other processes.
  - Has its own share of memory.
- A process' instructions are executed concurrently in multiple *threads*.
  - Threads share the memory of the process they belong to.
  - Threads communicate by writing and reading to memory.
- Threads can be executed on different processors (cores).
  - Processors do not write directly to main memory.
  - They use *caches* to improve read/write performance.
  - One processor cannot access the cache of another processor.
- Java threads are directly mapped to *OS threads*.
- Scala inherits the thread model of Java.



Costs of different CPU operations



http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles

Java Memory Model (JMM)

- Scala inherits the memory model from the JVM.
- The memory model defines *when writes to a variable are visible to other threads*.
- The compiler and the runtime perform various optimizations to gain performance.
  - *Registers* may be used as intermediate storage.
  - Data may be written to hierarchies of *caches*.
  - *Bytecode* statements may be *reordered*.
- The **rules** of the JMM define how threads interact through memory:
  - *Program order:* Program optimizations by the compiler must not alter the serial semantics within a thread.
  - *Locking:* Locks for the same synchronization variable must not overlap.
  - *Volatile fields:* A write to a volatile field is immediately visible to all threads.
  - *Thread start:* All actions in a thread are done after a call to start().
  - Many other rules.

Working with Threads in Scala

- Thread creation

```
class MyThread extends Thread:
  override def run(): Unit =
    println("Executed in new thread.")

val t = new MyThread
t.start()
```

- Convenience method to execute code in separate thread

```
def doInThread(body: => Unit): Thread =
  val t = new Thread:
    override def run() = body

  t.start()
  t
```

```
val t = doInThread { println("Executed in new thread.") }
```

Execution of Multi-Threaded Code

- If multiple threads access the same memory locations, the program behavior is not deterministic anymore.

```
var currId = 0L

def generateUniqueId() =
  val newId = currId + 1 // (1)
  currId = newId         // (2)
  newId

val t1 = doInThread { println(generateUniqueId()) }
val t2 = doInThread { println(generateUniqueId()) }
// Output: 1 2 or 2 1 or 1 1
```

  - The program works correctly if execution of statements (1) and (2) is not interfered by the other thread.
  - t1 and t2 generate the same id if both threads concurrently read the field *currId*.
- We say that there is a *race condition* in a program when the output of a program depends on the execution schedule of the instructions.

Synchronization

- Every Java/Scala object has a special property called an *intrinsic lock or monitor*.
- Only one thread can *acquire* ownership of a lock.
- If a thread tries to acquire the lock while another thread owns the lock, this thread is *blocked*.
- If a thread executes the statement `x.synchronized { statements }`
  - it tries to acquire the lock for **x** which is released at the end of the block.
  - The thread is blocked while **x** is locked by another thread.
- The race condition in the example can be removed by locking the region accessed by multiple threads:

```
var currId = 0L
def generateUniqueId() = this.synchronized
  val newId = currId + 1
  currId = newId
  newId
```

Data Races and Reordering

- How can this program result in `x==1 && y==1`?

```
var a = false; var b = false
var x = -1;    var y = -1
val t1 = doInThread { a = true; y = if (b) 0 else 1 }
val t2 = doInThread { b = true; x = if (a) 0 else 1 }
t1.join(); t2.join()
assert(!(x == 1 && y == 1))
```

  - It's required that a as well as b are `false` to produce `x==1 && y==1` as a result.
- Possible reasons for this "abnormal" behavior:
  - *Reordering of statements:* This reordering has no impact on serial semantics but influences behavior of concurrent execution.
  - *Data race:* Write of one thread (to cache) is not seen be the other thread.
- The `synchronized` keyword also guarantees that writes to memory of one thread are visible to all other threads.

Deadlocks

- Synchronization can result in deadlocks:

```
class Account(val name: String, var balance: Double);

def transfer(amount: Double, a: Account, b: Account): Unit =
  a.synchronized {
    b.synchronized { a.balance -= amount; b.balance += amount }
  }
```

```
var t1 = doInThread { for (i <- 0 until 1000)
                        transfer(10, account1, account2) }
var t2 = doInThread { for (i <- 0 until 1000)
                        transfer(10, account2, account1) }
```

- Deadlock can be avoided when resources are always locked in the same order:

```
def transfer(amount: Double, a1: Account, a2: Account): Unit =
  if (a1.name < a2.name)
    a1.synchronized { a2.synchronized { … } }
  else
    a2.synchronized { a1.synchronized { … } }
```

- In practice it's difficult to ensure a consistent ordering of resources.

Guarded Blocks

- Busy-waiting can be avoided by using Java's *wait/notify* mechanism.

```
class Consumer(val tasks: Queue[() => Unit]) extends Thread:
  def addTask(task: => Unit) = tasks.synchronized {
    tasks.enqueue(() => task)
    tasks.notifyAll();
  }
  private def getTask(): () => Unit = tasks.synchronized {
    while (tasks.isEmpty) tasks.wait() // guarded block
    tasks.dequeue();
  }
  override def run() =
    while true do
      val task = getTask()
      task()
```

- Wait can cause *spurious wakeups*: Condition thread is waiting for is not met.
- Condition must be checked repetitively → *guarded block*.

Volatile Variables

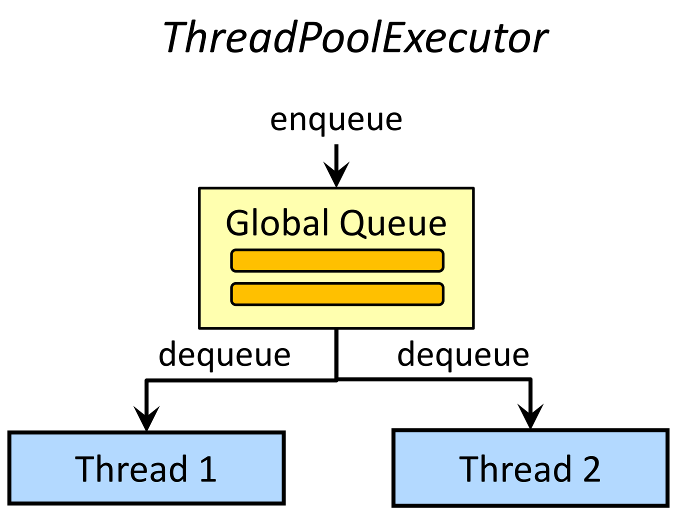
- Reads and writes to volatile variables are atomic.
- Operations (eg. incrementation) are not atomic.
- Writes to volatile variables are immediately visible to all threads.
  - Variables are not cached or held in registers.
  - Compiler will not reorder instructions.
- Thread communication via volatile variables is very fast.
- Most common application: status flags

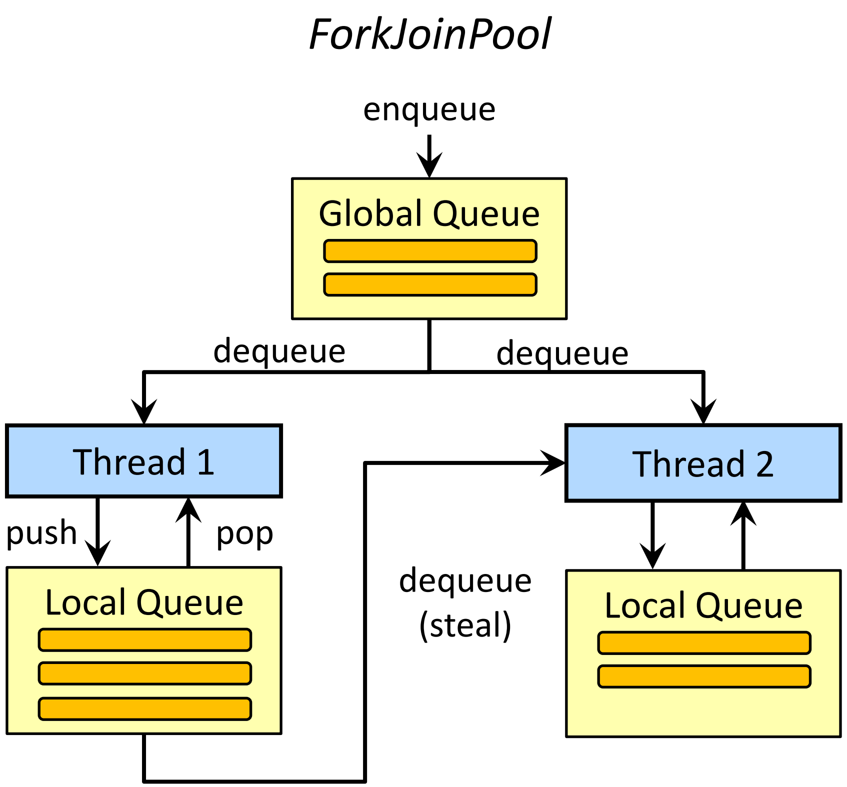
```
class Worker extends Thread:
  @volatile
  private var stopped = false
  override def run() =
    while !stopped do { /* do some work */ }
  def shutdown() = stopped = true
}
```

  - This program may not terminate if `@volatile` is omitted.

Thread Pools

- Creating processes is very expensive, creating threads is expensive.
- Thread pools manage a set of threads.



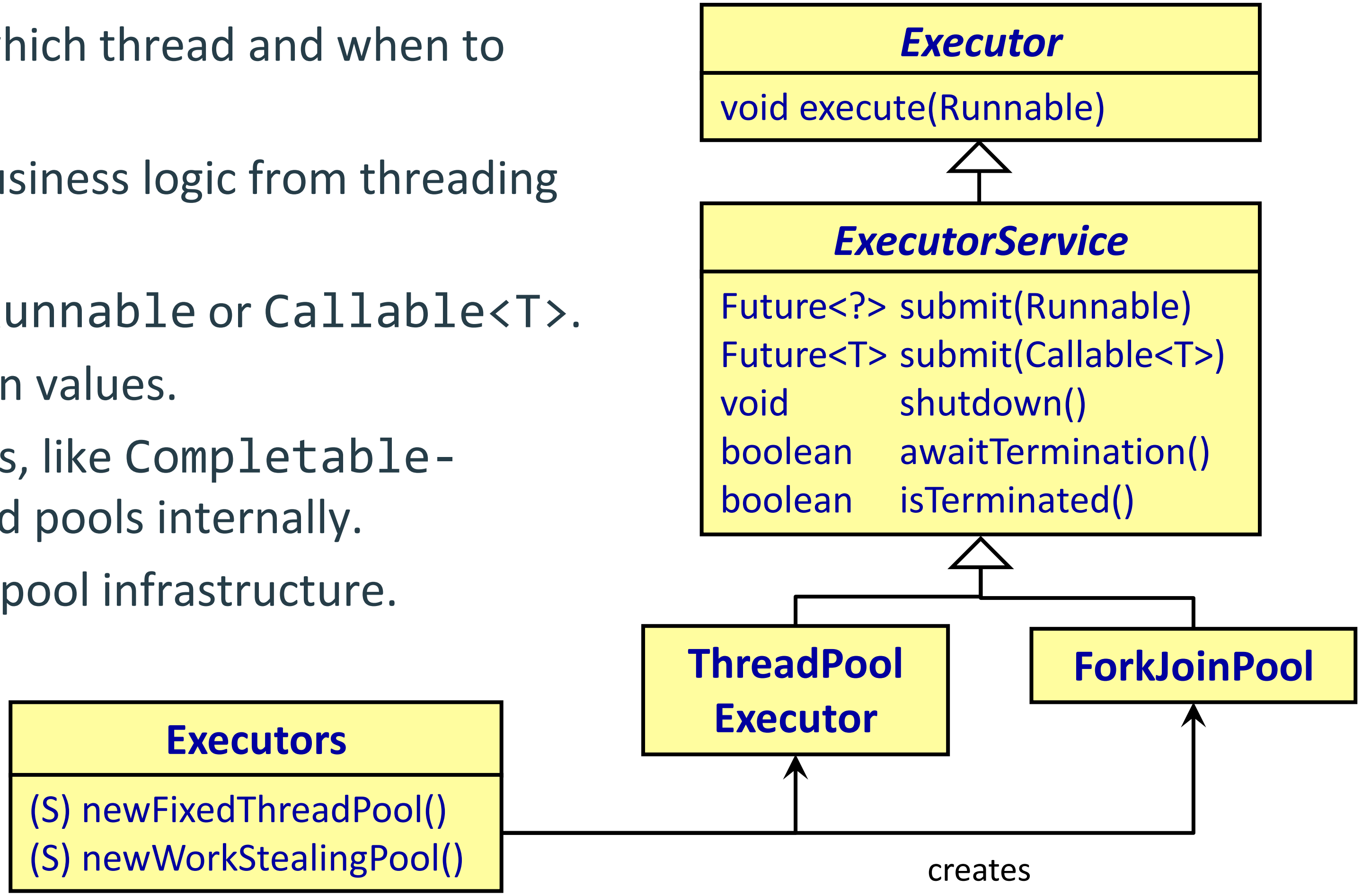


Advantages of *ForkJoinPool*

- Efficient for recursive tasks
- Less contention in local queue
- Load balancing achieved with *work stealing*

Executor and ExecutorService

- `Executor` decides on which thread and when to execute a task.
- `Executor` decouples business logic from threading infrastructure.
- Tasks must implement `Runnable` or `Callable<T>`.
- `Callable` objects return values.
- More advanced concepts, like `Completable-Futures` leverage thread pools internally.
- Scala uses Java's thread pool infrastructure.



Using Executors in Scala

- `Executor` implementations can be used directly in Scala.

```
val executor = new java.util.concurrent.ForkJoinPool
executor.execute(
  () => println("This task is run asynchronously."))
```

- `ExecutionContext` is similar, but more Scala specific.
- Internally `ExecutionContext` uses a `ForkJoinPool` instance.

```
val execCtx = scala.concurrent.ExecutionContext.global
execCtx.execute(
  () => println("This task is run asynchronously."))
```

  - Often an `ExecutionContext` object is passed implicitly to methods.

- A customized `ForkJoinPool` instance can also be used:

```
val execCtx = scala.concurrent.ExecutionContext.fromExecutorService(
                new java.util.concurrent.ForkJoinPool(2))
```

Atomic Variables

- Atomic variables support complex operations (more than one read/write), that are executed atomically.

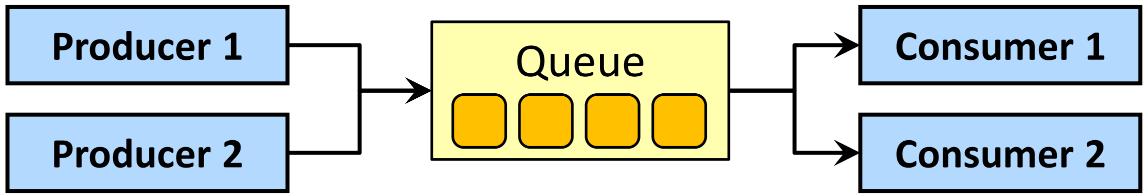
```
private val uid = new AtomicLong(0L)
def getUniqueId(): Long = uid.incrementAndGet()
```

- Operations are implemented *lock-free*
  - Operations are very fast
  - No danger of deadlocks
- Operations are implemented in terms of a fundamental atomic operation: *compareAndSet* (also called *compare-and-swap*).
  - Takes the expected previous value and the new value
  - Fails if previous value changed → operation has to be repeated

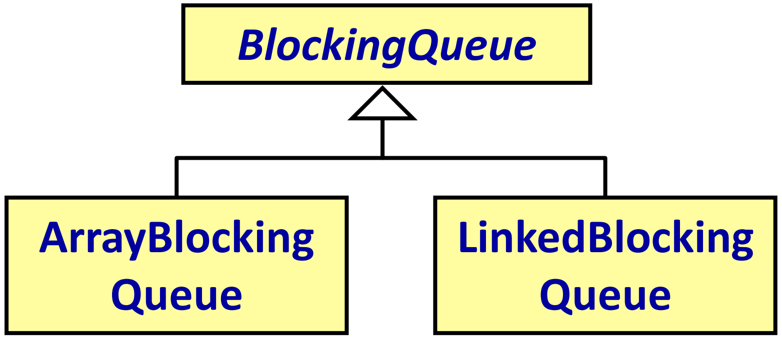
```
private val currId = new AtomicLong(0L)
@tailrec def generateUniqueId(): Long =
  val oldId = currId.get
  val newId = oldId + 1
  if currId.compareAndSet(oldId, newId) then newId else generateUniqueId()
```

Concurrent Collections

- *Producer-Consumer* is a common pattern in concurrent programming.
  - Work items need to be buffered in *concurrent queue*



- Java's concurrent queues can also be used in Scala.
- `BlockingQueue` supports three types of operations, depending on the behavior when queue is empty or full:
  - Operations throw an exception: `add/remove/element`
  - Operations return a special value: `offer/poll/peek`
  - Operations block: `put/take`



- Iterators are *weakly consistent*: modifications might or might not be reflected in iterator.

Foundations of Functional Programming

[Pilquist et al., 2023]

High-Order Functions

- Functions which take other functions as parameters are called *high-order functions*.
- It’s easy to create and to handle functions: Functions are first-class citizens in Scala.
- Example:

```
def sumSquares(a: Int, b: Int): Int =
  if a > b then 0 else a * a + sumSquares(a + 1, b)
def sumPowerOfTwo(a: Int, b: Int): Int =
  if a > b then 0 else Math.pow(2, a).toInt + sumPowerOfTwo(a + 1, b)
```

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
def sumSquares(a: Int, b: Int): Int = sum(x => x * x, a, b)
def sumPowerOfTwo(a: Int, b: Int): Int = sum(Math.pow(2, _).toInt, a, b)
```

- Parameterization of functions by other functions is a typical design pattern in functional programming.

Immutable State

- Typical procedural Java code:

```
public class Product { … }
public static List<String> outOfStock(List<Product> allProds) {
  List<Product> prods = new ArrayList<>();
  for (Product p : allProds) if (p.inStock < MIN_STOCK) prods.add(p);
  Collections.sort(prods, new Comparator<Product>() { … });
  List<String> prodNames = new ArrayList<String>();
  for (String p : prods) prodNames.add(p.getName());
  return prodNames;
}
```

- Functional Scala code is cleaner, because
  - no state has to be tracked,
  - code describes *what* has to be done and *not how* it has to be done.

```
case class Product(id: Int, name: String, inStock: Int)
def outOfStock(prods: Seq[Product]) = prods.filter(_.inStock < MIN_STOCK)
                                           .sortBy(_.id)
                                           .map(_.name)
```

Applying Functions to Collections

- `filter` composes a collection of all elements for which a given predicate holds:

```
val evenNumbers = List(1, 2, 3, 4, 5) filter (_ % 2 == 0) // List(2, 4)
```

- map applies a function to each element of a collection:

```
val lowerCase = List("a", "b", "c")
val upperCase = lowerCase map (_.toUpperCase) // List("A", "B", "C")
```

```
val lowerUpper = lowerCase map (str => List(str, str.toUpperCase))
  // List(List("a","A"), (List("b", "B"), List("c", "C"))
```

- `flatMap`
  - Applies a function that generates a sequence out of each element of a collection.
  - Flattens the resulting sequence of sequences to a (flat) sequence.

```
val lowerUpper = lowerCase flatMap (str => List(str, str.toUpperCase))
  // List("a", "A", "b", "B", "c", "C")
```

For Expressions (1)

- `map` and `flatMap` can be cascaded → code may be hard to understand:

```
case class Person(name: String, isMale: Boolean, children: Seq[Person])
val persons: Seq[Person] = List(adam, eve, cain, abel)
val parentChildPairs = persons flatMap (p =>
                         p.children map (c => (p.name, c.name)))
```

  - yields in `List((Adam,Cain), (Eve,Cain), (Eve,Abel))`
  - using `map` would yield in `List(List((Adam,Cain)), …, List())`
- The following for expression does exactly the same:

```
val parentChildPairs =
  for (p <- persons;
       c <- p.children)
    yield (p.name, c.name)
```

  - The compiler translates the second query into the first one.
  - The second query is much clearer to read.

For Expressions (2)

- Filters

```
val motherChildPairs = persons.filter (p => !p.isMale)
                         flatMap (p =>
                           p.children map (c => (p.name, c.name)))
```

  can also be mapped to for expressions

```
val motherChildPairs =
  for (p <- persons if (!p.isMale);
       c <- p.children) yield (p.name, c.name)
```

- For expressions with side effects

```
for (p <- persons) println(p.name)
```

  are mapped to the function `foreach`

```
persons.foreach (p => println(p.name));
```

- A for expression can be applied to all domains that provide `map`, `flatMap`, `filter`/ `withFilter` (and `foreach`).

Monads

- There are many types that define `map` and `flatMap`.
- A type `M` that supports the three operations `flatMap`, `map`, and `unit` is called a monad:

```
trait M[T]:
  def flatMap[U](f: T => M[U]): M[U]
  def map[U]    (f: T => U   ): M[U]

def unit[T](x: T): M[T]
```

- `map` can be defined in terms of `flatMap` and `unit`:

```
def map[U](f: T => U) : M[U] = m flatMap (x => unit(f(x)))
```

- In a stricter definition monads must fulfill certain *laws*.
- In other functional languages `flatMap` is called `bind`.
- A monad need not be a collection. Often monads handle a *single value*.
- In Scala the name of the `unit` function is monad-specific.

The Option Monad

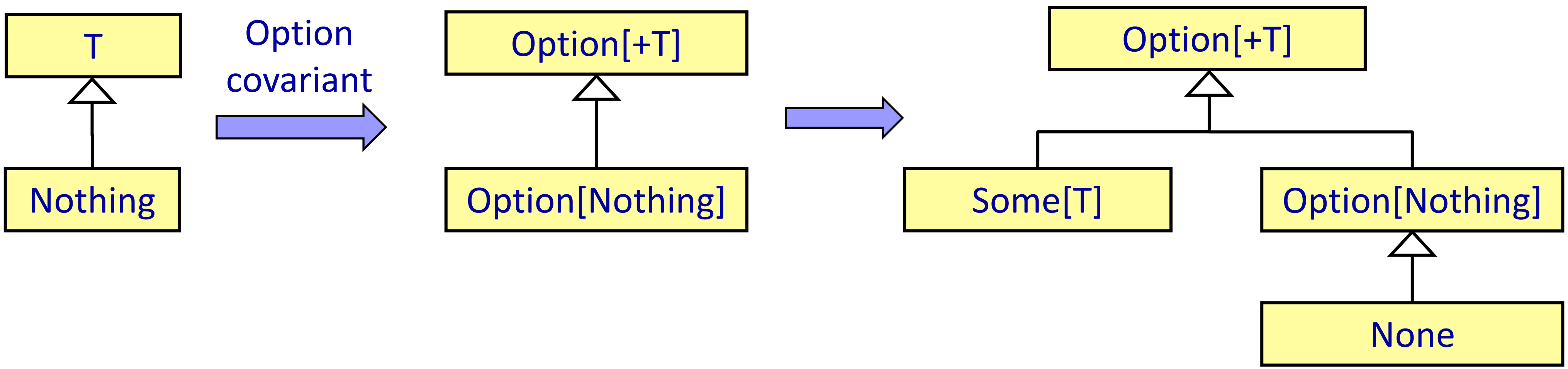
- Options instances hold values of two forms:
  - `Some(x)` where x is the actual value
  - None represents an undefined value

```
abstract class Option[+T]:
  def flatMap[U](f: T => Option[U]): Option[U] = this match
    case Some(x) => f(x)
    case None => None
  def map[U](f: T => U): Option[U] = this match
    case Some(x) => Some(f(x))
    case None => None

case class Some[+T](element : T) extends Option[T]
case object None                 extends Option[Nothing]
```

- Advantages over `null`
  - Cleaner interfaces: It's obvious that value may be undefined.
  - Checking for undefined value is enforced.

The Option Monad: Embedding into the Type System



- Application:

```
val op1: Option[String] = Some("Hello")
val op2: Option[String] = None
```

The Option Monad: map

- The unit operations Some an None are generators for monad values:

```
val o1: Option[Double] = Some(9)
Val o2: Option[Double] = None
```

- map transforms Option values to other Option values:

```
def toDouble(value: String): Double = value.toDouble

Some("9") map (n => toDouble(n)) // → Some(9.0)
None      map (n => toDouble(n)) // → None
```

- map allows for *function composition*:

```
def sqrt(value: Double): Double = math.sqrt(value)
Some("9") map (n => toDouble(n)) map (n => sqrt(n)) // → Some(3.0)
```

- Applying a function to a None value does not result in an exception:

```
None map (n => toDouble(n)) map (n => sqrt(n)) // → None
```

The Option Monad: `flatMap`

- `map` is not always useful when applied to option values because it generates “nested objects”:

```
def toDouble(value : String) : Option[Double] =
  try   { Some(value.toDouble) }
  catch { case nfe : NumberFormatException => None }

Some("9") map (n => toDouble(n)) // → Some(Some(9.0))
Some("x") map (n => toDouble(n)) // → Some(None)
```

- This can be prevented by applying `flatMap`:

```
Some("9") flatMap (n => toDouble(n)) // → Some(9.0)
Some("x") flatMap (n => toDouble(n)) // → None
```

- In this manner functions can also be composed:

```
def sqrt(value : Double) : Option[Double] = value match { … }

Some("9")  flatMap (n => toDouble(n)) flatMap (n => sqrt(n)) // → Some(3.0)
Some("x")  flatMap (n => toDouble(n)) flatMap (n => sqrt(n)) // → None
Some("-9") flatMap (n => toDouble(n)) flatMap (n => sqrt(n)) // → None
```

Monad Laws

- Associativity

```
(m flatMap f) flatMap g == m flatMap (x => f(x) flatMap g)
```

```
(Some("4") flatMap (toDouble(_))) flatMap (sqrt(_)) ==
  Some("4") flatMap (x => toDouble(x) flatMap (sqrt(_))) == Some(2.0)
(None flatMap (toDouble(_))) flatMap (sqrt(_)) ==
  None flatMap (x => toDouble(x) flatMap (sqrt(_))) == None
```

- Left unit

```
unit(x) flatMap f == f(x)
```

```
Some("1") flatMap (toDouble(_)) == toDouble("1") == Some(1.0)
```

- Right unit

```
m flatMap unit == m
```

```
Some("1") flatMap (Some(_)) == Some("1")
None flatMap (Some(_)) == None
```

Monads and For Expressions

- Because monads support `map` and `flatMap` they can be used in for expressions.
- Function composition expressions like

```
val res = Some("4") flatMap (toDouble(_))
                      flatMap (sqrt(_))
```

can also be written as for expressions:

```
val res =
  for (o1 <- Some("4");
       o2 <- toDouble(o1);
       o3 <- sqrt(o2)) yield o3;
```

- Advantage: better readability

The Try Monad (1)

- Try can have two types of values:
  - `Success(x)`: Computation ended successfully with result x.
  - `Failure(ex)`: Computation failed with Exception ex.

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

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

  - Fatal exceptions: `OutOfMemoryError`, `StackOverflowError`, `InterruptedException`, etc.
- Application:
  - Composition of functions that may fail
  - Pass results between functions, threads and processes

The Try Monad (2)

- Like each monad `Try` must define `flatMap` and `map`:

```
abstract class Try[+T]:
  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)) // Try.apply(f(x))
    case fail: Failure => fail
```

- In the stricter interpretation `Try` is not a monad because the *left unit law does not hold*:

```
Try(expr) flatMap f != f(expr)
```

  - The left-hand side never throws a (non-fatal) exception whereas the right-hand side will raise any exception thrown by `f`.

The Try Monad: map

- The unit operations `Success` and `Failure` are generators for monad values:

```
val success: Try[String] = Success("9")
val failure: Try[String] = Failure(IllegalArgumentException())
```

- `map` transforms `Try` values to other `Try` values:

```
def toDouble1(value: String): Double = value.toDouble

Success("9") map (n => toDouble1(n)) // → Success(9.0)
Failure(ex)  map (n => toDouble1(n)) // → Failure(ex)
```

- `map` allows for *function composition*:

```
def sqrt1(value: Double): Double = value match
  case v: Double if v >= 0 => math.sqrt(value)
  case _                   => throw IllegalArgumentException()

Success("9") map (n => toDouble1(n)) map (n => sqrt1(n)) // → Success(3.0)
```

- Applying a function to a `Failure` value does not result in an exception:

```
Failure(ex) map (n => toDouble1(n)) map (n => sqrt1(n)) // → Failure(ex)
```

The `Try` Monad: `flatMap`

- `map` is not always useful when applied to `Try` values because it generates “nested objects”:

```
def toDouble2(value: String): Try[Double] = Try { value.toDouble }

Success("9") map (n => toDouble2(n)) // → Success(Success(9.0))
Success("x") map (n => toDouble2(n)) // → Success(Failure(NumberFormatException))
```

- This can be prevented by applying `flatMap`:

```
Success("9") flatMap (n => toDouble2(n)) // → Success(9.0)
Success("x") flatMap (n => toDouble2(n)) // → Failure(NumberFormatException)
```

- In this manner functions can also be composed:

```
def sqrt2(value: Double): Try[Double] = value match
  case v : Double if v>=0 => Success(math.sqrt(value))
  case _                  => Failure(IllegalArgumentException())

Success("9")  flatMap (n => toDouble2(n)) flatMap (n => sqrt2(n))
                                          // → Success(3.0)
Success("x")  flatMap (n => toDouble2(n)) flatMap (n => sqrt2(n))
                                          // → Failure(NumberFormatException)
Success("-9") flatMap (n => toDouble2(n)) flatMap (n => sqrt2(n))
                                          // → Failure(IllegalArgumentException)
```

The Try Monad: For Expressions

- Because `Try` supports `map` and `flatMap`, `Try` can be used in for expressions:

```
val r1 = for (d <- toDouble2("9")) yield sqrt1(d)
           // → Success(3.0)
val r2 = for (d <-toDouble2("9"); s <- sqrt2(d)) yield s
           // → Success(3.0)
val r3 = for (d <-("x"); s <- sqrt2(d)) yield s
           // → Failure(NumberFormatException)
val r4 = for (d <- toDouble2(" 9"); s <- sqrt2(d)) yield s
           // → Failure(IllegalArgumentException)
```

- `Try` objects can be processed by pattern matching:

```
r1 match
  case Success(x)  => println(s"Success($x)")
  case Failure(ex) => println(s"Failure($ex)")
```

Futures and Promises

[Prokopec, 2017; p. 101 ff.]

Asynchronous Programming

- In certain situations, threads have to wait for external resources (e. g. data arriving over the network).
- Traditional programs *block* threads in such cases.
- Possible problems with *blocking synchronization*:
  - Poor utilization of threads
  - Danger of deadlocks
  - Starvation of threads pools
- Asynchronous programming:
  - Thread has to wait for resource → separate computation is scheduled.
  - Thread continues.
  - Original computation is continued when resource is available.
  - Prevents expensive context switches.
- Futures and Promises are abstractions that support asynchronous programming.

Futures

- A future

```
trait Future[T]:
  def value: Option[Try[T]]
object Future:
  def apply[T](computation: => T): Future[T]
```

  - is a placeholder for a value of type T → *future value*
  - executes an asynchronous computation → *future computation*
- Starting a future computation:

```
val f : Future[Int] = Future {
  computeNthPrime(1000)
} // shorthand for Future.apply(…)
```

- Accessing the future value (never do this in this way in production code):

```
println(f.value) // → None
while (! f.isCompleted) { Thread.sleep(100) }
println(f.value) // → Some(Success(7919))
```

ExectionContext

- Future computations are carried out in an `ExecutionContext`.
- The `ExecutionContext` is passed as the 2nd parameter to `Future.apply`:

```
object Future:
  def apply[T](computation :=>T)
              (using executor: ExecutionContext): Future[T]
```

- The Scala library provides a default implementation for this implicit parameter:

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

- The global `ExecutionContext` is backed by `ForkJoinPool`
  - The actual/minimum/maximum number of threads is set to `Runtime.availableProcessors`.
- The `ExectionContext` can be easily replaced:

```
given ExecutionContext =
  ExecutionContext.fromExecutor(Executors.newFixedThreadPool(10))
```

Future callbacks (1)

- Callbacks are (partial) functions that are called when the future computation succeeds or fails.

```
trait Future[T]:
  def foreach(f: T => U)              // success
  def failed: Future[Throwable]       // Failure(ex) → Success(ex)
  def onComplete(f: Try[T] => Unit)   // success or failure
```

  - Futures callbacks are called *eventually* after the computation is finished and independently of other callbacks.

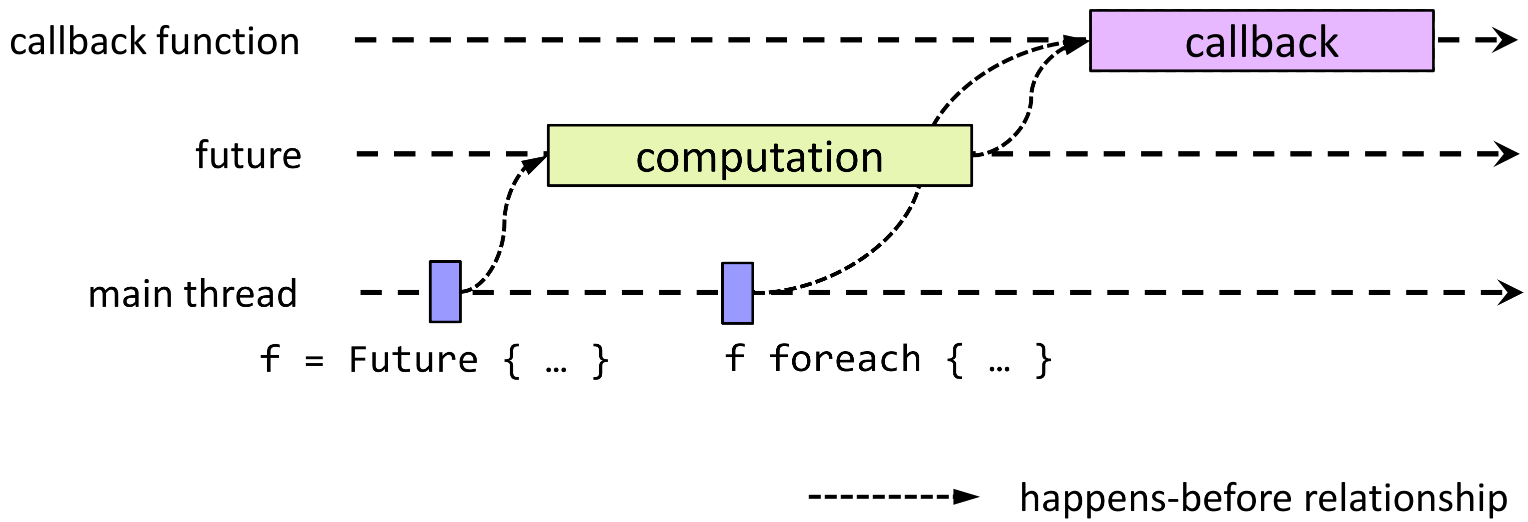
- Example:

```
val f: Future[List[Int]]= Future { getPrimes(100) }
f foreach { primes => println(primes.mkString(", ")) }
f.failed foreach { ex => println(s"Failed with $ex") }
f onComplete {
  case Success(primes) => println(primes.mkString(", "))
  case Failure(ex)     => println(s"Failed with $ex")
}
```

Future callbacks (2)

- Installing a callback is a non-blocking operation (like future creation).
- Callback may or may not be executed in a separate thread.

```
val f = Future { computation }
f foreach { callback }
```



Functional Composition of Futures (1)

- Problems related to callbacks
  - Cascading futures becomes cumbersome (nested callbacks)
  - Combining the results of multiple futures is difficult
  - Error handling becomes confusing
- Functional composition is a pattern in which functions are cascaded using high-order functions, so called *combinators*.
- Monads provide combinators and are therefore an important means for functional composition.

$$\text{monad}_1 \xrightarrow{\text{combinator}_1(f_1)} \text{monad}_2 \xrightarrow{\text{combinator}_2(f_2)} \text{monad}_3$$

- Futures support combinators like `map`, `flatMap`, `filter`, etc. that can be used to build pipelines of future computations.

$$\text{future}_1 \xrightarrow{\text{flatMap}(f_1)} \text{future}_2 \xrightarrow{\text{map}(f_2)} \text{future}_3$$

Functional Composition of Futures (2)

- Combinators generate futures out of futures → combinators don't block
- Combinators provided by `Future`:

```
trait Future[T]:
  def map[S]    (f: T => S)          : Future[S]
  def flatMap[S](f: T => Future[S]) : Future[S]
  def filter    (p: T => Boolean)    : Future[T]
  def recover[U >: T](pf: PartialFunction[Throwable, U]) : Future[U]
```

  - `map`: Apply function `f` to future value. `f` is executed in a new future. Exceptions are propagated to the new future.
  - `flatMap`: Similar to map, but result is not nested
    - `val f1: Future[Int] = Future { 1 };`
`val f2 = f1 map { i => Future { i+1 } }` → `f2: Future[Future[Int]]`
    - `val f1: Future[Int] = Future { 1 };`
`val f2 = f1 flatMap { i => Future { i+1 } }` → `f2: Future[Int]`
  - `recover`: Function `pf` handles exceptions thrown in future computations. Valid results will be propagated to the new future.

Functional Composition of Futures (3)

- Example for `recover`:

```
val f1 = Future (6 / 0) recover { case e: ArithmeticException => 0  }
           → Success(0)
val f2 = Future (6 / 0) recover { case e: NullPointerException => 0 }
           → Failure(ArithmeticException)
val f3 = Future (6 / 2) recover { case e: ArithmeticException => 0  }
           → Success(3)
```

- Example for `map`:

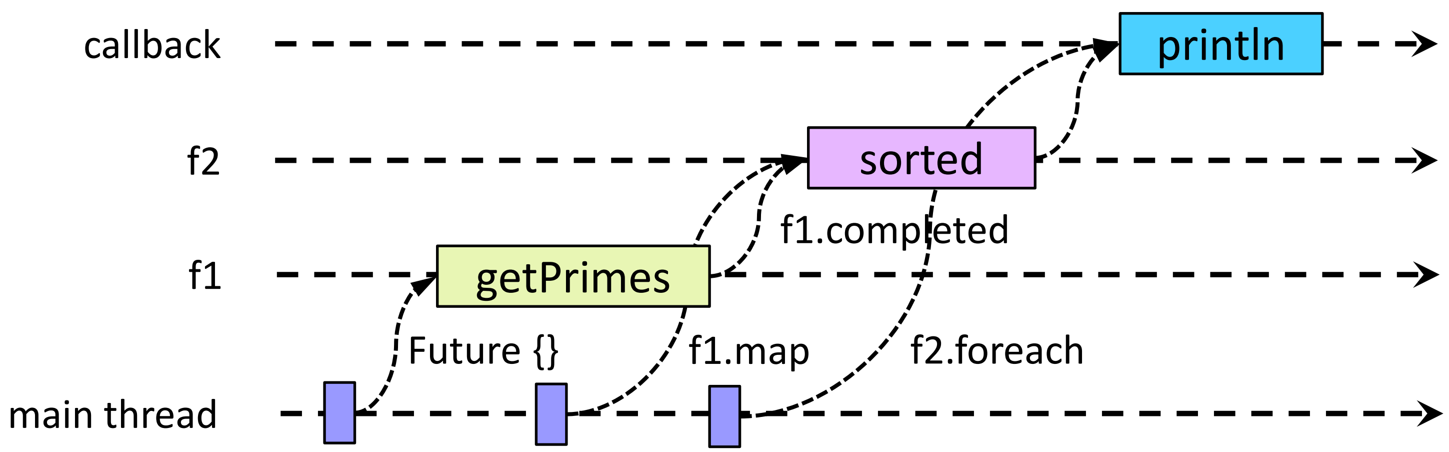
```
val f1: Future[List[Int]]= Future { getPrimes(100) }
val f2: Future[List[Int]] = f1 map { primes => primes.sorted }
f2 foreach { primes => println(primes.mkString(", ")) }
```

- Example for `flatMap`:

```
val f1 = Future { getPrimes(2, 50) }
val f2 = Future { getPrimes(51, 100) }
val f3 = f2 flatMap { primes2 => f1 map { primes1 => primes1 ++ primes2 } }
val f4 = f3 map { primes => primes.sorted }
f4 foreach { primes => println(primes.mkString(", ")) }
```

Semantics of the map Combinator

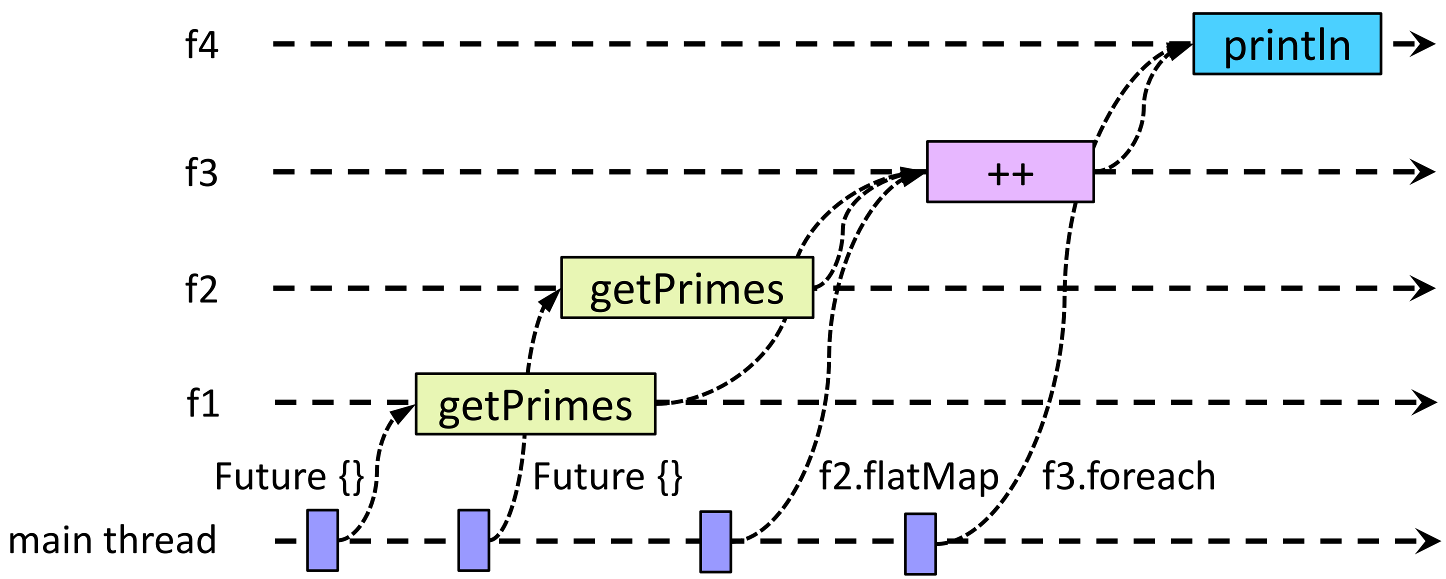
```
val f1: Future[List[Int]]= Future { getPrimes(100) }
val f2: Future[List[Int]] = f1 map { primes => primes.sorted }
f2 foreach { primes => println(primes.mkString(", ")) }
```



Futures and combinators may (but need not) be executed in separate threads.

Semantics of the `flatMap` Combinator (1)

```scala
val f1 = Future { getPrimes(2, 50) }
val f2 = Future { getPrimes(51, 100) }
val f3 = f2 flatMap { primes2 => f1 map { primes1 => primes1 ++ primes2 } }
f3 foreach { primes => println(primes.mkString(", ")) }
```



Semantics of the `flatMap` Combinator (2)

- Difference between `Future.flatMap` and `Future.map`:
  - Let `f: Future[U]`, `g: U => V`
  - `val res = f flatMap { u => Future { g(u) } }`
    - Execution order:
      - Future `f` completes with a future value of type U.
      - `Future { g(u) }` completes with a future value of type V.
      - Future `res` completes with a future value of type V.
    - Return type: `Future[V]`.
  - `val res = f map { u => Future { g(u) } }`
    - Execution order:
      - Future `f` completes,
      - Future `res` completes with a future value of type `Future[V]`,
      - `Future { g(u) }` completes.
    - Return type: `Future[Future[V]]`.

Futures and For Expressions

- Since futures support the methods `map`, `flatMap`, `foreach` (and `withFilter`), futures can be combined in for expressions.
- Therefore, this program fragment

```
val f1 = Future { getPrimes(2, 50) }
val f2 = Future { getPrimes(51, 100) }
val f3 = f2 flatMap { primes2 => f1 map { primes1 => primes1 ++ primes2 } }
f3 foreach { primes => println(primes.mkString(", ")) }
```

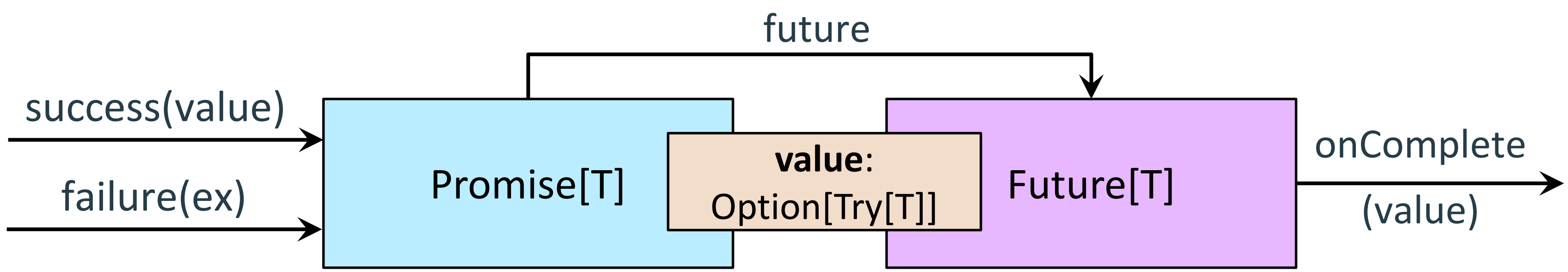
can be rewritten to

```
val f1 = Future { getPrimes(2, 50) }
val f2 = Future { getPrimes(51, 100) }
val f3 = for (primes1 <- f1;
              primes2 <- f2) yield primes1 ++ primes2
for (primes <- f3) println(primes.mkString(", "))
```

- Advantage: Program structure is easier to understand.

Promises (1)

- Futures and promises represent two aspects of a *single-assignment variable*:
  - The *promise* allows you to assign a value to the future.
  - The *future* allows you to read the value.



- Example usage of promises:

```
val p = Promise[String]()
p.future onComplete {
  case Success(v)  => println(v)
  case Failure(ex) => println(ex)
}
p success "Hello"
// p.failure new Exception("failed")
```

Promises (2)

- A new value can only be *assigned once* to a future.
  - `success`, `failed`, and `complete` throw an exception if one tries to override a value.
  - `trySuccess`, `tryFailed`, and `tryComplete` return a value to indicate that the assignment was successful.
- It's easy to implement futures by means of promises:

```
import ExecutionContext.Implicits.global

def doInFuture[T](computation: => T): Future[T] =
  val p = Promise[T]()
  global.execute(() =>
    try
      val result = computation
      p success result
    catch
      case NonFatal(e) => p failure e

  p.future
```

Future-Callback Bridge

- The control flow of callback-based APIs is not apparent from the code.
- The API dictates the control flow → *inversion of control*.
- Promises can be used to bridge the gap between callback-based APIs and futures.

```
case class Event(value: Any)
@FunctionalInterface
trait EventListener:
  def onEvent(evt: Event)
```

```
class EventSource:
  def addEventListener(l: EventListener) = …
```

```
def eventOccurred(): Future[Event] =
  val p = Promise[Event]()
  val source = new EventSource
  source.addEventListener((evt: Event) => p trySuccess evt)
  p.future
```

```
eventOccurred() foreach { evt => println(s"event [$evt] occurred.") }
```

Futures and Blocking

- If all threads of the thread pool are blocked, no further future computation can be started.

```
var futures = for (_ <- 1 to noProcessors)
                yield Future { Thread.sleep(1000) }
val f = Future { … }
```

  - Computation of future `f` will only start after 1000 ms.
- This situation is called *thread starvation*: shared resources are made unavailable for long periods by greedy threads.
- Avoid blocking whenever possible.
- If blocking is absolutely necessary, enclose code in a `blocking` call.

```
var futures = for (_ <- 1 to noProcessors)
                yield Future { blocking { Thread.sleep(1000) } }
val f = Future { … }
```

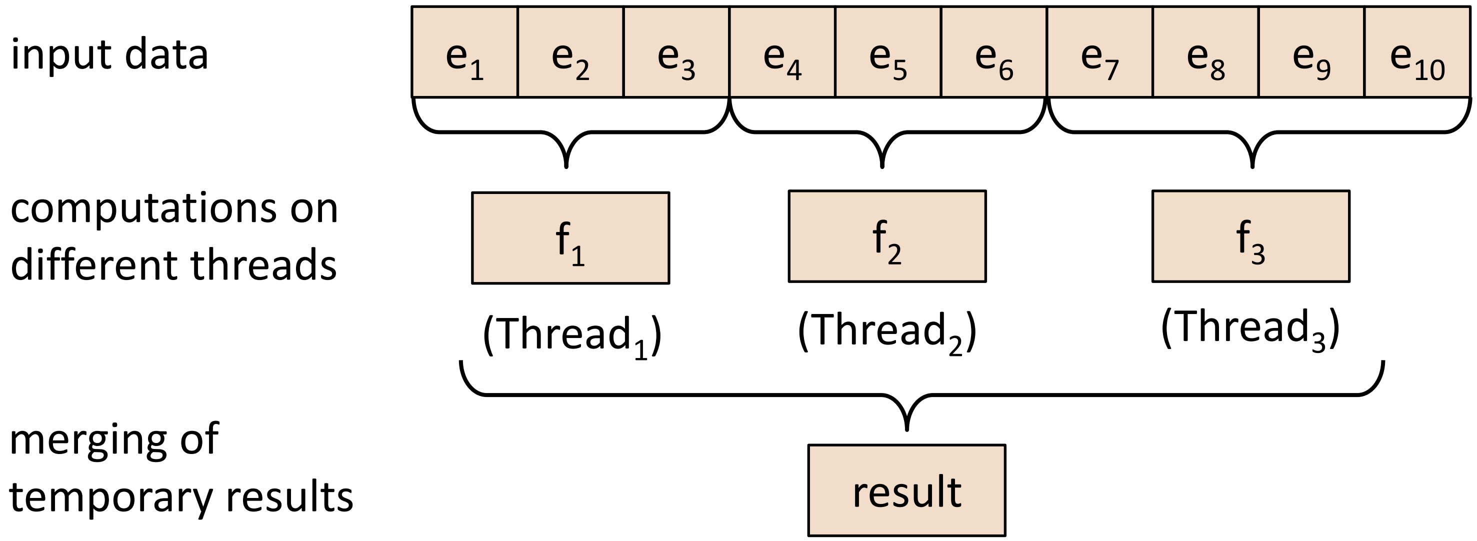
  - The execution context will provide additional worker threads if necessary.

Data Parallel Collections

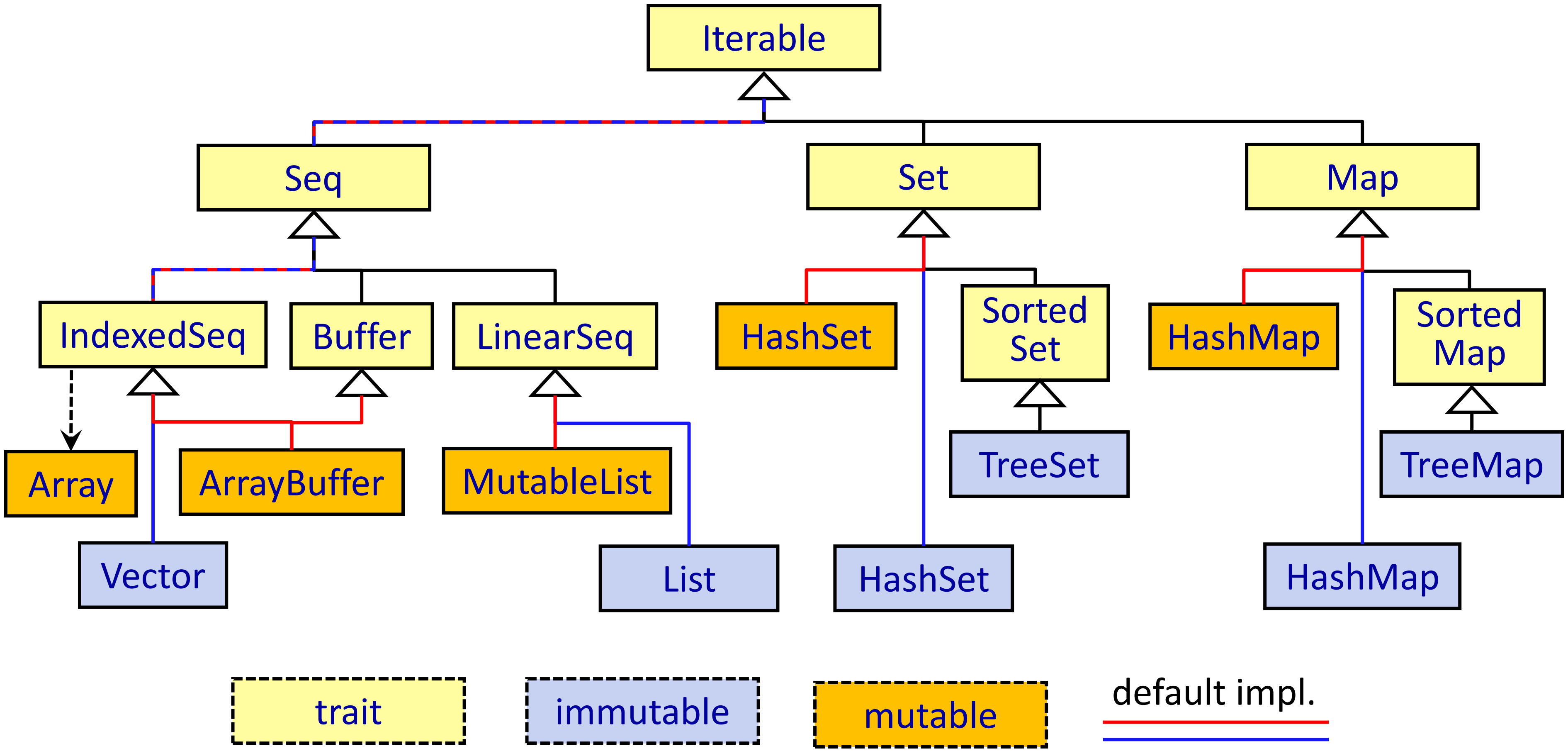
[Prokopec, 2017; p. 137 ff.]

Data Parallelism

- There are two forms of parallelism
  - **Task parallelism** (control parallelism) focusses on distributed tasks which are performed across different threads and processors.
  - **Data parallelism** focusses on performing the same operations on different data elements.
- Basic principle of data parallelism



Scala's Collection Hierarchy



Semantics of Scala Collections

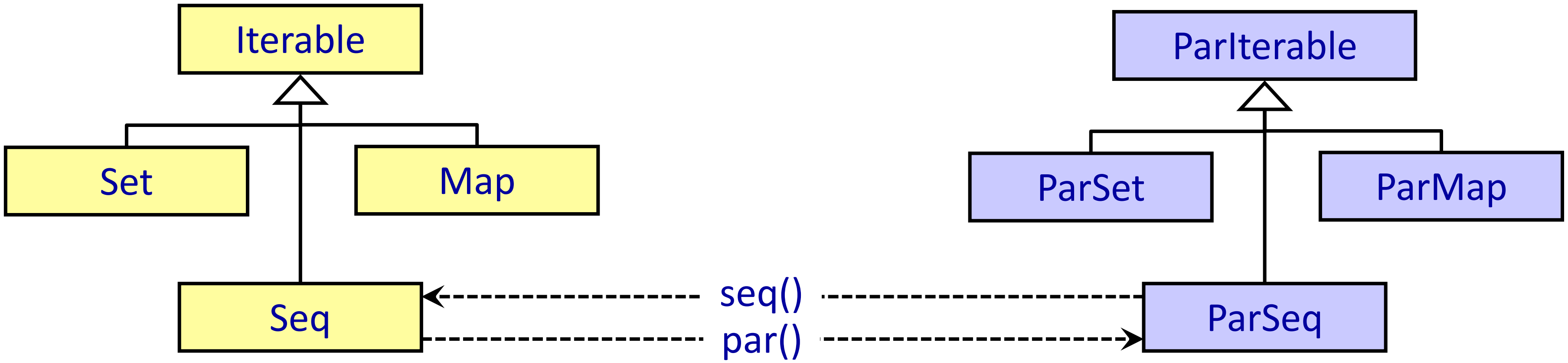
- **`Iterable`**:
  - Trait that implements behavior common to all collections based on the high-order function `foreach.`
  - Provides an abstract method that generates an `Iterator`.

```
def foreach[U](f: Elem => U): Unit
def iterator: Iterator[T]
```

  - Iterator defines abstract methods `hasNext` and `next`.
- **`Seq`**: Provides a method `apply` for indexing (numerical index).
- **`Buffer`**: Used to create sequences by appending (`+=`), prepending (`+=:`), inserting, updating, or removing elements.
- **`LinearSeq`**: Efficient implementation of `head`, `tail`, `isEmpty`. Most other operations are linear.
- Many types provide an `apply` method in the companion object that should be used to create collection instances (default implementation will be used).

```
val seq = Seq(1,2,3)                 // Vector
val mutSeq = mutable.Seq(1, 3, 2) // ArrayBuffer
```

Parallel Collections



- The method `par` converts a sequential collection into its parallel counterpart.
- Methods of parallel collections like `map`, `foreach`, `filter`, `max`, `sum`, etc. execute in parallel.
- Example:

```
def seqSumSquares(seq: Seq[Int]) = seq.map(i => i*i).sum
def parSumSquares(seq: ParSeq[Int]) = seq.map(i => i*i).sum
val numbers = 1 to 1000000
val seqSum = seqSumSquares(numbers)
val parSum = parSumSquares(numbers.par)
```

Specifics of Parallel Collection

- Parallelizable collections
  - Parallel *splitters* recursively split a collection into smaller parts:

    ```
    trait Splitter[+T] extends Iterator[T]:
      def split: Seq[Splitter[T]]
    ```

  - The asymptotic running time of `split` must be O(log n). Otherwise, the recursive division would produce too much overhead.
  - *Parallelizable collections:* `Array`, `Vector`, `HashSet`, `HashMap`, etc.
  - *Non-parallelizable collections:* `List`, `Stream`
- Side effects in parallel operations

```
var sum = new AtomicLong(0)
numbers.par foreach { i => sum += i }
```

  - When multiple threads use variables, access must be synchronized, or atomic variables must be used.
  - Parallel collection operation operations must be associative.