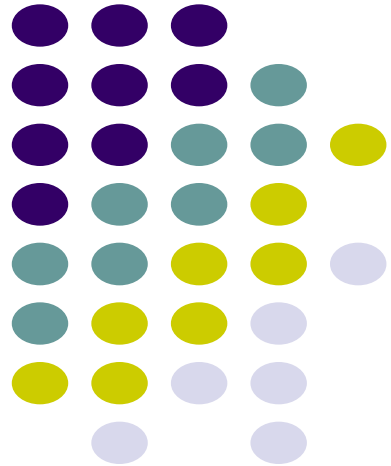
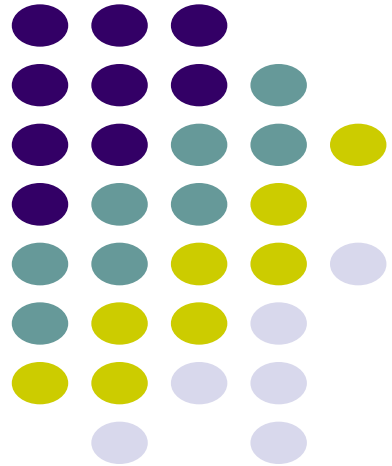
Course
Parallel Programming and Distributed Systems in Java

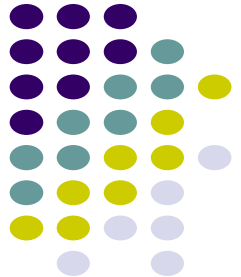


Лекция 5

Actors Model of Computation

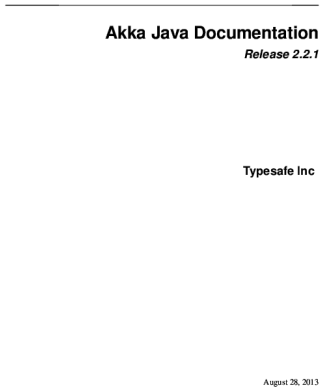


Цель

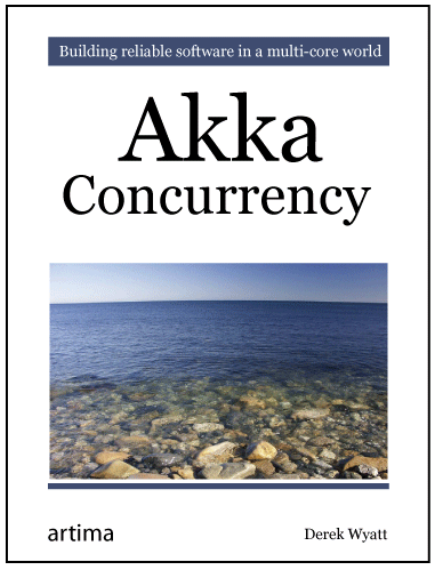


- Ознакомиться с основами модели акторов
  - Private mutable state
  - Immutable messages
- Ознакомиться с библиотекой Akka
  - UntypedActors
  - Mailboxes
  - Dispatchers

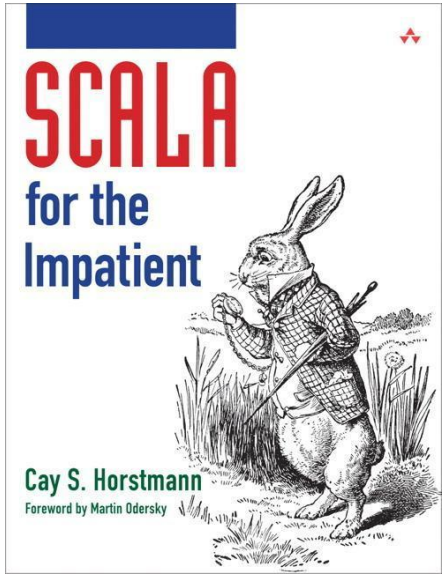
Что почитать?



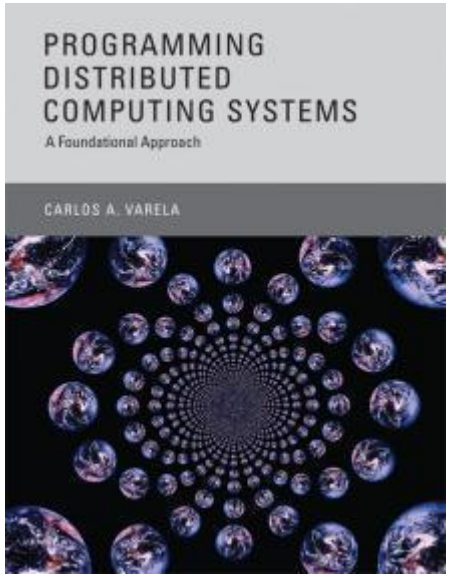
[Official Docs]



by Derek Wyatt,
May 2013



by Cay Horstmann,
March 2012



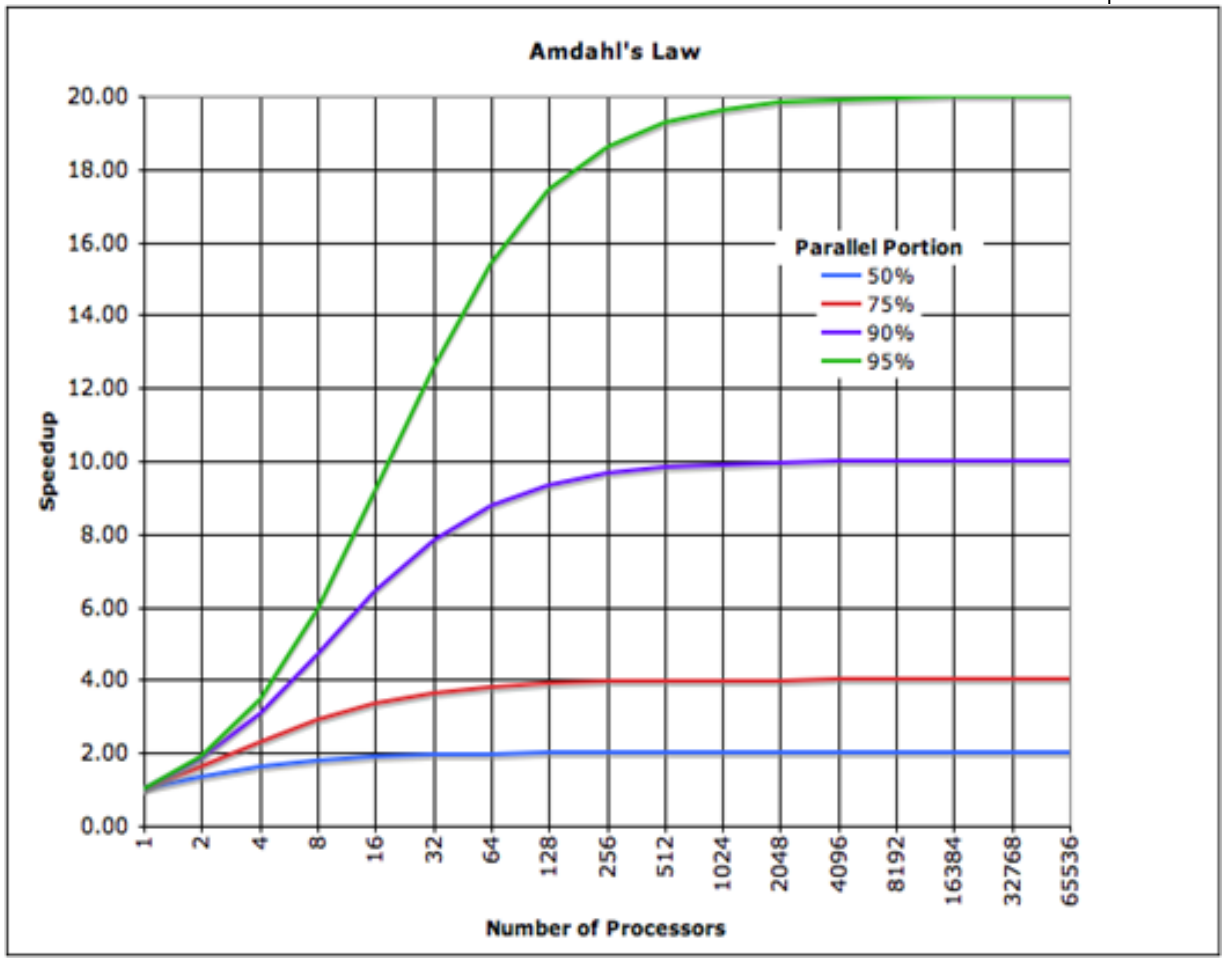
by Carlos Varela,
May 2013

Кто уже использует?

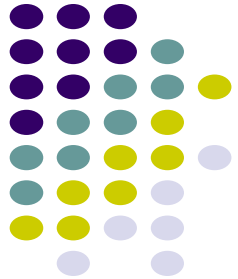
at&t

Краткое повторение

- Частоты не увеличиваются с 2006 года
- Закон Мура все еще работает, но для количества ядер, а не частоты

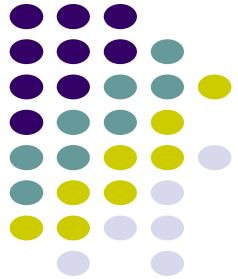


Parallelism and Concurrency



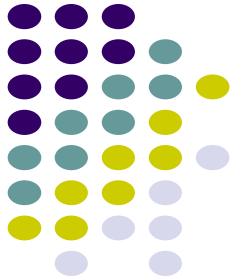
- Concurrency – возникает когда хотя бы 2 потока делают вычисление используя общий ресурс
- Parallelism – возникает когда хотя бы 2 потока выполняются одновременно
- Оба подхода сложны, т.к. включают в себя общее состояние

Shared Memory



- Race Conditions
  - Вспомните свою первую лабораторную. Вы наверняка замечали что результат бывает непредсказуем
- Dead locks

Решения

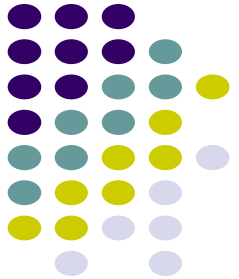


- Функциональное программирование – все объекты неизменяемые (immutable)
  - scala> List(1,2,3).par.map(_ + 2)

    res: List[Int] = List(3,4,5)

- Actors – держать изменяемое состояние изолированным и обмениваться друг с другом неизменяемыми сообщениями

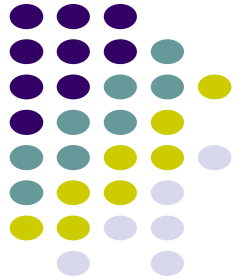
История (Actors)



- Формальное определение модели акторов дал Карл Хьюитт в 1973

- Впервые использовалась компанией Ericsson в середине 80х
  - Изобретение языка Erlang
  - Построение отказоустойчивой системы OTP с доступностью **99.9999999%** (nine nines)

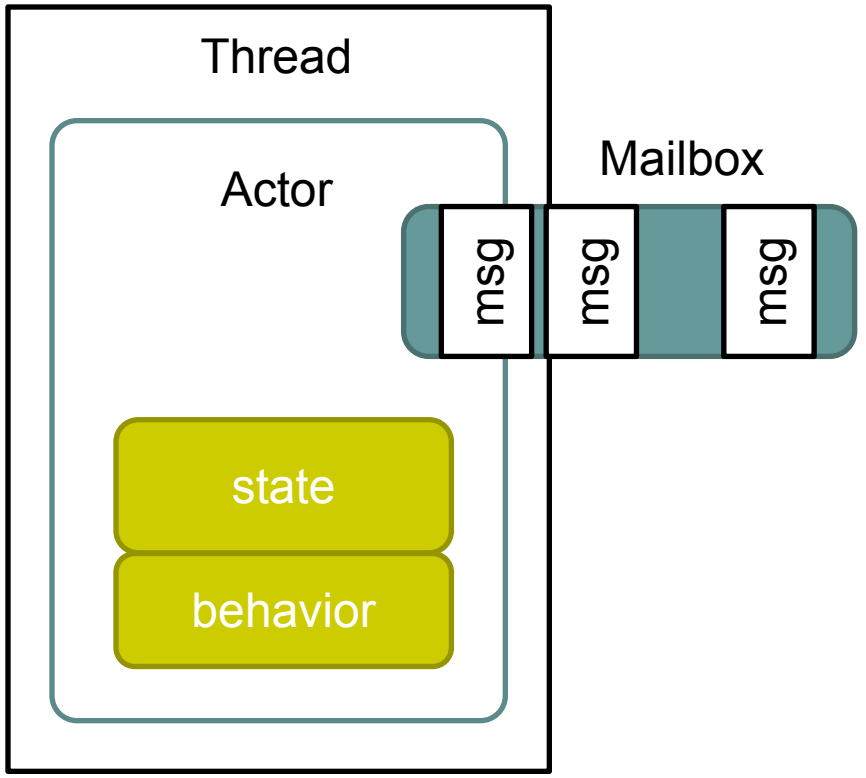
История (AkkA)



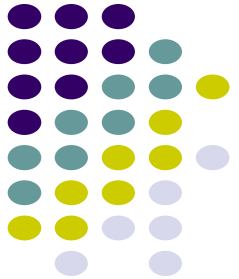
- Первая версия – 2009 г.
- Написана на Scala – объектно-функциональном языке
  - Работает поверх JVM
  - Scala может вызывать Java код
  - Java может вызывать Scala код – с ограничениями

Актор

- Легковесный объект
- Выполняется в собственном потоке
- Не использует общее состояние
- Сообщения в ящике (mailbox) с сохранением порядка
- Малый отклик из-за короткого call-стека

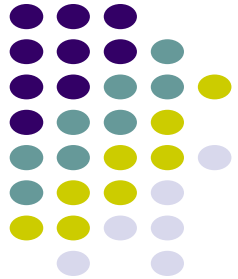


Актор может



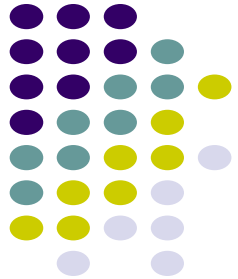
- Содержать изменяемые значения
- Обращаться только к своим изменяемым значениям
  - AkkA гарантирует это.
  - Не предоставляет прямой доступ к Актору
  - Клиенты работают только с прокси объектами – ActorRef
- Обрабатывать сообщения от других Акторов (Mailbox)
- Отправлять сообщения другим Акторам (ActorRef)
- Порождать новых Акторов

Составляющие Актора (Mailbox)



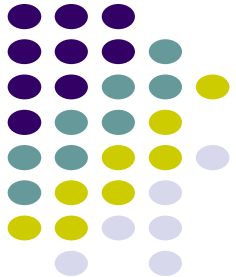
- Mailbox – каждый Актор содержит очередь и ожидает сообщения в эту очередь
- Сообщения неизменяемые (immutable)
  - нет Race Conditions и проблем с видимостью и очередностью (Visibility and Reordering)

Составляющие Актора (Dispatcher)



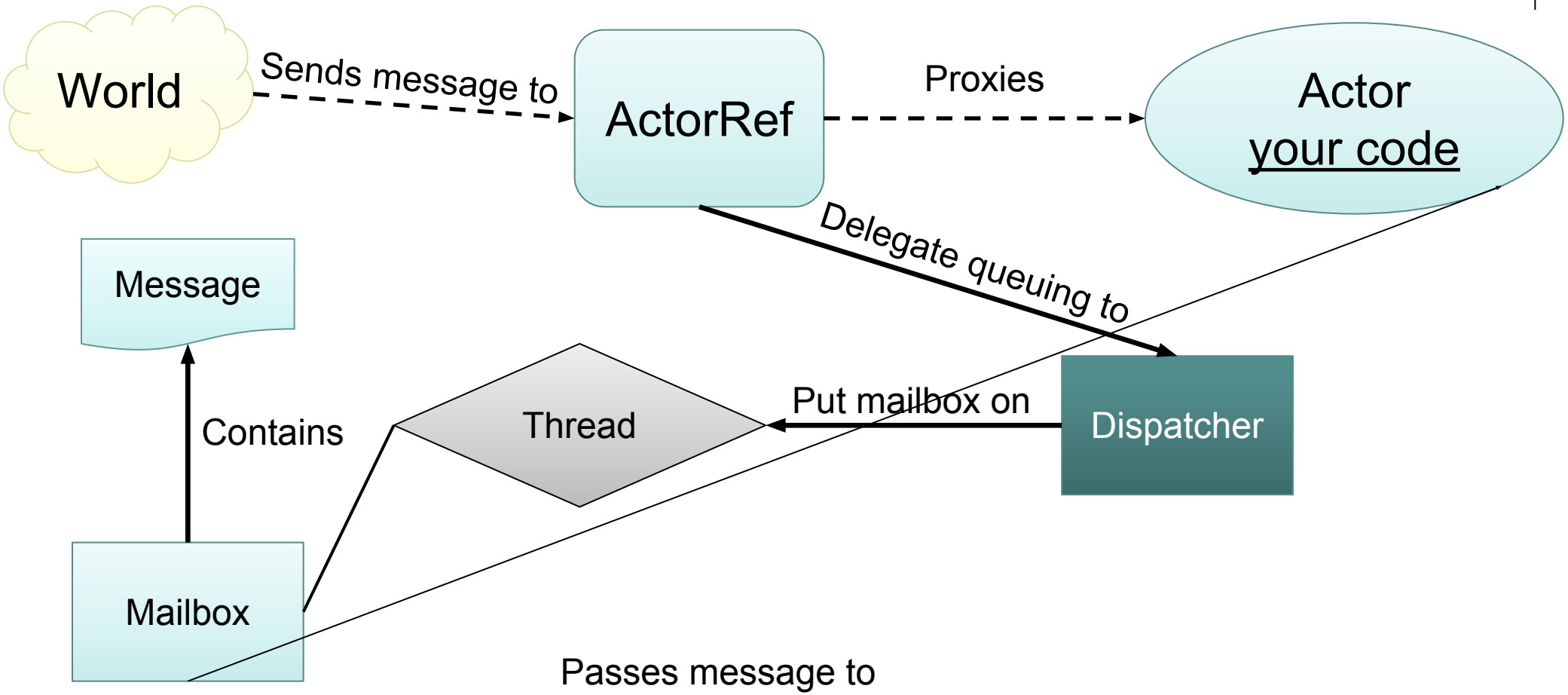
- Actor != Thread
  - Лучше считать что это Task
- Как отделить выполнение задачи от потока?
  - Создать пул на 20 потоков и раздать 100 задач на обработку этим пулом
- Аналогично с Акторами
  - X Акторов обрабатываются Y потоками
  - X >> Y
- Dispatcher – отвечает за доставку сообщения целевому актору, стоящему за ActorRef

Составляющие Актора (Other)

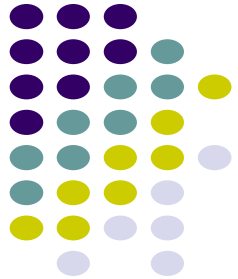


- Actor – объект, реализующий метод receive. Вызывается когда mailbox получает сообщение
- ActorRef – прокси объект Актора. Через него адресуются сообщения
- ActorSystem – конфигурирует и связывает все элементы вместе. Точка входа. Корневой «Актор»

Составляющие Актора

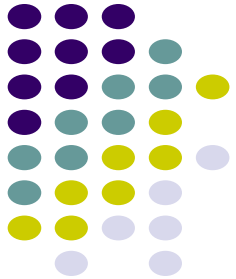


Объявление Акторов



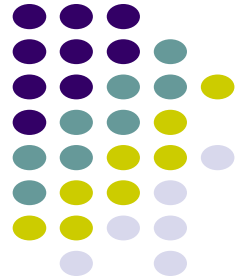
- Простейший Актор – это класс унаследованный от UntypedActor в пакете akka.actor
- Нужно определить один метод:
  - public void onReceive(final Object message)
- Этот метод определяет как обрабатывать сообщения

Простой актор



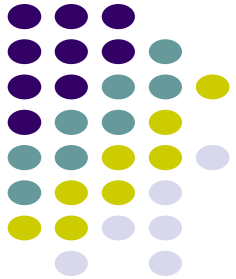
```
public class Counter extends UntypedActor {
   private int count = 0;
   public void onReceive(final Object message) {
      if (message instanceof Interger) {
         count += (Integer) message;
         System.out.println(“Count: “ + count);
      }
   }
}
```

Простой Актор – Scala Teaser



```
class Counter extends Actor {
  var count = 0
  def receive = {
    case n: Int => {count += n; println(count)}
  }
}
```

Создание Акторов



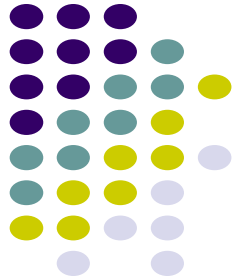
- Для создания используется метод `actorOf`
  - Аргументом является класс создаваемого актора
  - Результат – `ActorRef`, ссылается на созданный актор

```
final ActorRef counter = system.actorOf(Counter.class);
counter.start();
```

- Через полученную ссылку можно отправлять сообщения

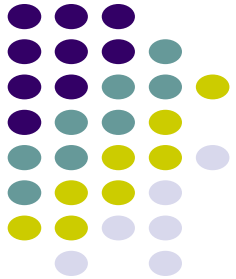
```
counter.tell(new Integer(100))
```

Отправка сообщений



- 2 способа
  - `tell(final Object msg)` – асинхронно отправить сообщение
  - `Future ask(final Object msg)` – отправить сообщение, и получить `Future`
- Не путать с `java.util.concurrent.Future`, но принцип тот-же
  - `future.await()` – блокирует выполнение до результата
  - Затем `future.result().get()` – чтобы получить ответ, отправленный другим актором.
  - `Future.result().isDefined()` – чтобы проверить, существует-ли этот ответ

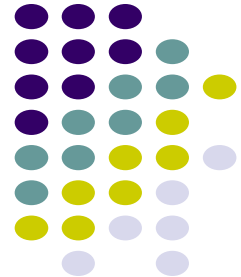
Ответ на сообщение



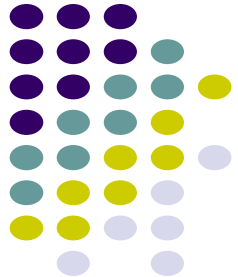
- Когда получаете сообщение от другого актора:
  - внутри метода `onReceive()` – можно получить ссылку на отправителя с помощью `getSender()`
  - получив `ActorRef` - используйте `tell()` или `ask()`

- Ответ на сообщение == отправка нового сообщения

DEMO

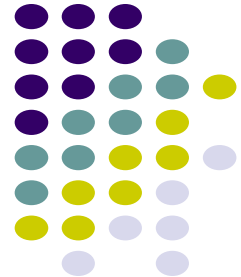


Смена поведения. HotSwap

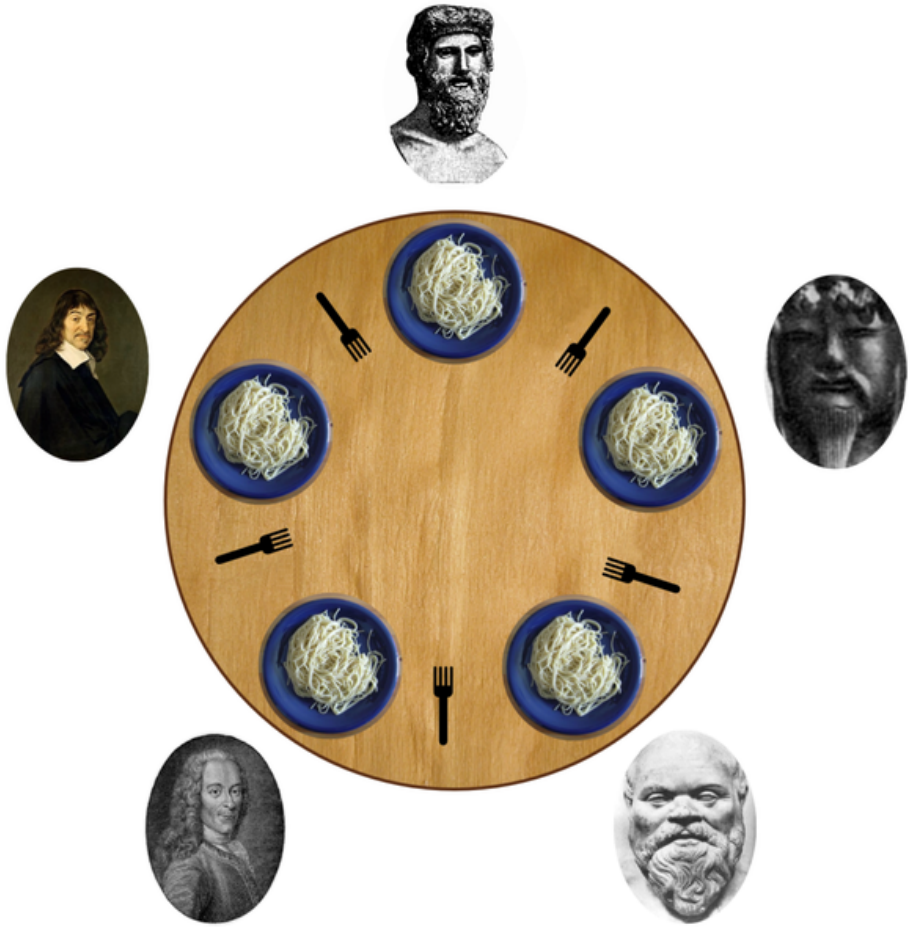


- Актор может изменить реализацию метода `onReceive` на лету
- `void become(Procedure behavior, Boolean replaceOld)`
- `void unbecome()`

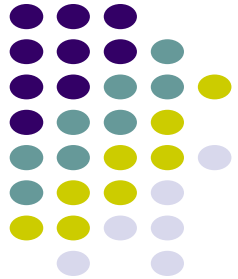
DEMO



Обедающие философы



Isolated Mutability



- Это принцип проектирования:
  - Содержит изменяемое состояние
  - Но гарантирует, что доступ к этому состоянию имеет только один поток
- Модель Акторов – реализует Isolated Mutability
  - Содержит множество легковесных потоков – Акторов
  - Каждый Актор содержит любое количество изменяемых переменных
    - Потому что никто кроме того Актора не имеет к ним доступа