NWEN303 Concurrent Programming

19: Model solution for Ass4,
and a new ConcurrentRunner

Marco Servetto
VUW

Abstracting over actors

In Ass4 we have seen how to have generalized actors that can serve certain roles, in this case, a producer of T that stockpiles up to a certain amount of products.

It is not -**any actor**-

but is also not -**a specific actor**-, like Alice.

- Can we imagine more of those actor roles?

Producer

```java
public abstract class Producer<T> extends AbstractActor{
  public static final class GiveOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private GiveOne() {};//prevent construction
    public static final GiveOne instance=new GiveOne();}

  private static final class MakeOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private MakeOne() {};//prevent construction
    public static final MakeOne instance=new MakeOne();}

  private final int maxSize;
  private final Class<T> product;
  private List<T> s = new ArrayList<>();
  private boolean running = false;
  public Producer(int max,Class<T> p) {this.maxSize = max; this.product = p;}

  protected abstract CompletableFuture<T> make();

  private void giveOne(GiveOne g) {/*..in a few slides..*/}
  private void makeOne(MakeOne g) {/*..in a few slides..*/}

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiveOne.class,this::giveOne)
      .match(MakeOne.class,this::makeOne)
      .match(product,this.s::add)
      .build();
    }  }
```

Producer

```java
public abstract class Producer<T> extends AbstractActor{
  public static final class GiveOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private GiveOne() {};//prevent construction
    public static final GiveOne instance=new GiveOne();}

  private static final class MakeOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private MakeOne() {};//prevent construction
    public static final MakeOne instance=new MakeOne();}

  private final int maxSize;
  private final Class<T> product;
  private List<T> s = new ArrayList<>();
  private boolean running = false;
  public Producer(int max,Class<T> p) {this.maxSize = max; this.product = p;}

  protected abstract CompletableFuture<T> make();

  private void giveOne(GiveOne g) {/*..in a few slides..*/}
  private void makeOne(MakeOne g) {/*..in a few slides..*/}

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiveOne.class,this::giveOne)
      .match(MakeOne.class,this::makeOne)
      .match(product,this.s::add)
      .build();
    }  }
```

Producer

```
public abstract class Producer<T> extends AbstractActor{
  public static final class GiveOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private GiveOne() {};//prevent construction
    public static final GiveOne instance=new GiveOne();}

  private static final class MakeOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private MakeOne() {};//prevent construction
    public static final MakeOne instance=new MakeOne();}

  private final int maxSize;
  private final Class<T> product;
  private List<T> s = new ArrayList<>();
  private boolean running = false;
  public Producer(int max,Class<T> p) {this.maxSize = max; this.product = p;}

  protected abstract CompletableFuture<T> make();

  private void giveOne(GiveOne g) {/*..in a few slides..*/}
  private void makeOne(MakeOne g) {/*..in a few slides..*/}

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiveOne.class,this::giveOne)
      .match(MakeOne.class,this::makeOne)
      .match(product,this.s::add)
      .build();
    }  }
```

Producer

```
public abstract class Producer<T> extends AbstractActor{
  public static final class GiveOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private GiveOne() {};//prevent construction
    public static final GiveOne instance=new GiveOne();}

  private static final class MakeOne implements Serializable{//singleton pattern
    private static final long serialVersionUID = 1L;
    private MakeOne() {};//prevent construction
    public static final MakeOne instance=new MakeOne();}

  private final int maxSize;
  private final Class<T> product;
  private List<T> s = new ArrayList<>();
  private boolean running = false;
  public Producer(int max,Class<T> p) {this.maxSize = max; this.product = p;}

  protected abstract CompletableFuture<T> make();

  private void giveOne(GiveOne g) {/*..in a few slides..*/}
  private void makeOne(MakeOne g) {/*..in a few slides..*/}

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiveOne.class,this::giveOne)
      .match(MakeOne.class,this::makeOne)
      .match(product,this.s::add)
      .build();
    }  }
```

Producer

```
public abstract class Producer<T> extends AbstractActor{
  ...
  private void giveOne(GiveOne g) {
    if(!s.isEmpty()){
      sender().tell(s.remove(s.size()-1),self());
      }
    else{
      pipe(make(), context().dispatcher()).to(sender());
      }
    //additionally: (here we sort of know the list is not full)
    if(running){return;}//when we are not running
    running=true;
    self().tell(MakeOne.instance,self());
    }

  private void makeOne(MakeOne g) {
    assert running;
    if(s.size()>=maxSize) {running=false;return;}//anyway we stop if list is full
    CompletableFuture<T> made = make();
    pipe(made, context().dispatcher()).to(self());
    made.thenAccept(o->self().tell(MakeOne.instance,self()));
    }
```

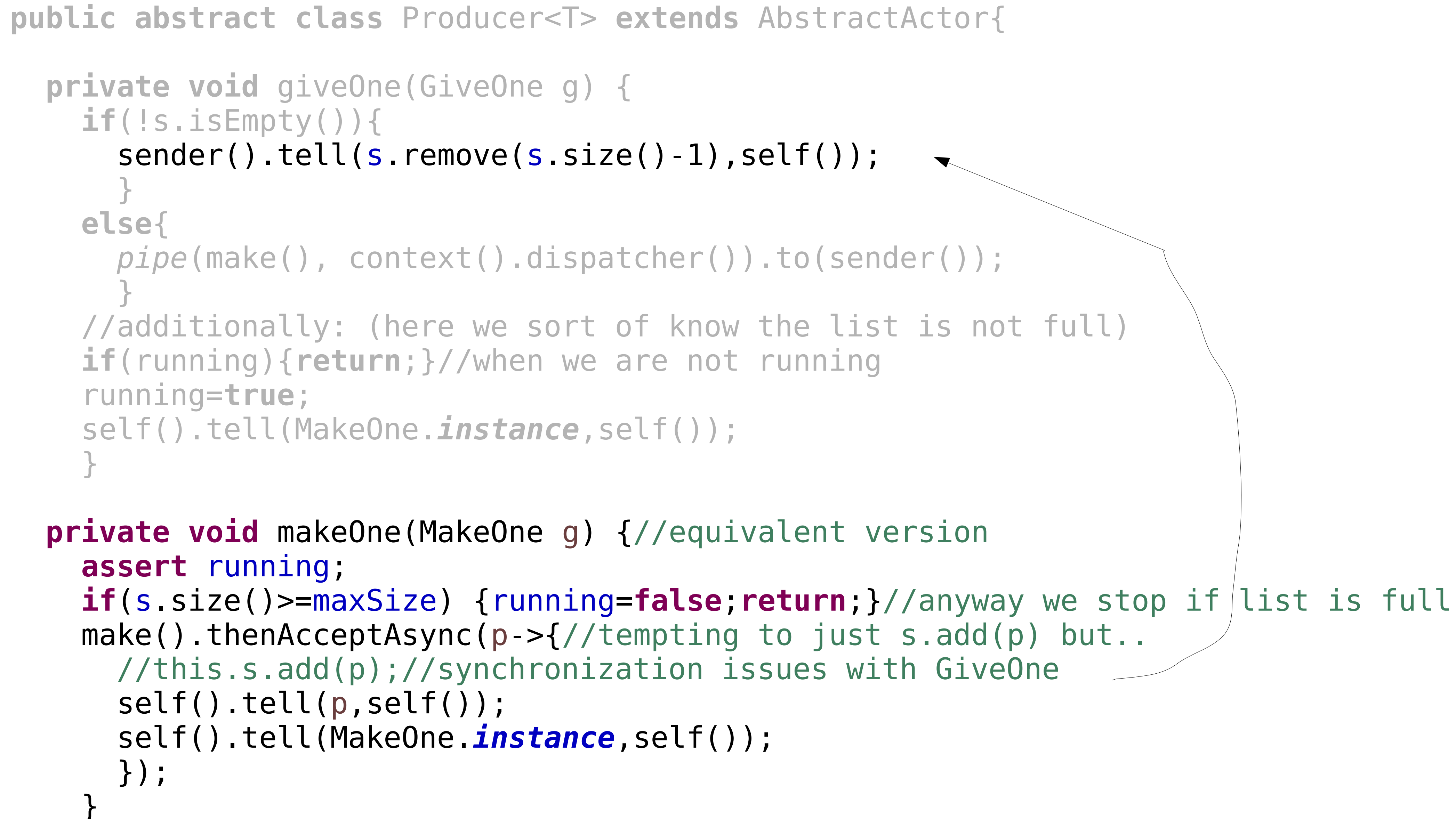
Producer

```
public abstract class Producer<T> extends AbstractActor{

  private void giveOne(GiveOne g) {
    if(!s.isEmpty()){
      sender().tell(s.remove(s.size()-1),self());
      }
    else{
      pipe(make(), context().dispatcher()).to(sender());
      }
    //additionally: (here we sort of know the list is not full)
    if(running){return;}//when we are not running
    running=true;
    self().tell(MakeOne.instance,self());
    }

  private void makeOne(MakeOne g) {
    assert running;
    if(s.size()>=maxSize) {running=false;return;}//anyway we stop if list is full
    CompletableFuture<T> made = make();
    pipe(made, context().dispatcher()).to(self());
    made.thenAccept(o->self().tell(MakeOne.instance,self()));
    }
```

Producer

```
public abstract class Producer<T> extends AbstractActor{

  private void giveOne(GiveOne g) {
    if(!s.isEmpty()){
      sender().tell(s.remove(s.size()-1),self());
      }
    else{
      pipe(make(), context().dispatcher()).to(sender());
      }
    //additionally: (here we sort of know the list is not full)
    if(running){return;}//when we are not running
    running=true;
    self().tell(MakeOne.instance,self());
    }

  private void makeOne(MakeOne g) {
    assert running;
    if(s.size()>=maxSize) {running=false;return;}//anyway we stop if list is full
    CompletableFuture<T> made = make();
    pipe(made, context().dispatcher()).to(self());
    made.thenAccept(o->self().tell(MakeOne.instance,self()));
    }
```

Producer



Our Producers

```
class Alice extends Producer<Wheat>{
  public Alice(int maxSize) {super(maxSize,Wheat.class);}
  public CompletableFuture<Wheat> make() {
    return CompletableFuture.supplyAsync(()->new Wheat());}
  }

class Bob extends Producer<Sugar>{
  public Bob(int maxSize) {super(maxSize,Sugar.class);}
  public CompletableFuture<Sugar> make() {
    return CompletableFuture.supplyAsync(()->new Sugar());}
  }

class Charles extends Producer<Cake>{
  ActorRef alice; ActorRef bob;
  public Charles(ActorRef alice,ActorRef bob,int maxSize) {
    super(maxSize,Cake.class); this.alice=alice; this.bob=bob;}

  public CompletableFuture<Cake> make() {
    CompletableFuture<Object> wheat=Patterns.ask(alice, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    CompletableFuture<Object> sugar=Patterns.ask(bob, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    return CompletableFuture.allOf(wheat,sugar).thenApply(v->{
      Sugar s = (Sugar) sugar.join();
      Wheat w = (Wheat) wheat.join();
      return new Cake(s, w);
      });
    }}
```

Our Producers

```
class Alice extends Producer<Wheat>{
  public Alice(int maxSize) {super(maxSize,Wheat.class);}
  public CompletableFuture<Wheat> make() {
    return CompletableFuture.supplyAsync(()->new Wheat());}
  }

class Bob extends Producer<Sugar>{
  public Bob(int maxSize) {super(maxSize,Sugar.class);}
  public CompletableFuture<Sugar> make() {
    return CompletableFuture.supplyAsync(()->new Sugar());}
  }

class Charles extends Producer<Cake>{
  ActorRef alice; ActorRef bob;
  public Charles(ActorRef alice,ActorRef bob,int maxSize) {
    super(maxSize,Cake.class); this.alice=alice; this.bob=bob;}

  public CompletableFuture<Cake> make() {
    CompletableFuture<Object> wheat=Patterns.ask(alice, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    CompletableFuture<Object> sugar=Patterns.ask(bob, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    return CompletableFuture.allOf(wheat,sugar).thenApply(v->{
      Sugar s = (Sugar) sugar.join();
      Wheat w = (Wheat) wheat.join();
      return new Cake(s, w);
      });
    }}
```

Our Producers

```
class Alice extends Producer<Wheat>{
  public Alice(int maxSize) {super(maxSize,Wheat.class);}
  public CompletableFuture<Wheat> make() {
    return CompletableFuture.supplyAsync(()->new Wheat());}
  }

class Bob extends Producer<Sugar>{
  public Bob(int maxSize) {super(maxSize,Sugar.class);}
  public CompletableFuture<Sugar> make() {
    return CompletableFuture.supplyAsync(()->new Sugar());}
  }

class Charles extends Producer<Cake>{
  ActorRef alice; ActorRef bob;
  public Charles(ActorRef alice,ActorRef bob,int maxSize) {
    super(maxSize,Cake.class); this.alice=alice; this.bob=bob;}

  public CompletableFuture<Cake> make() {
    CompletableFuture<Object> wheat=Patterns.ask(alice, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    CompletableFuture<Object> sugar=Patterns.ask(bob, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    return CompletableFuture.allOf(wheat,sugar).thenApply(v->{
      Sugar s = (Sugar) sugar.join();
      Wheat w = (Wheat) wheat.join();
      return new Cake(s, w);
      });
    }}
```

Our Producers

```java
class Alice extends Producer<Wheat>{
  public Alice(int maxSize) {super(maxSize,Wheat.class);}
  public CompletableFuture<Wheat> make() {
    return CompletableFuture.supplyAsync(()->new Wheat());}
  }

class Bob extends Producer<Sugar>{
  public Bob(int maxSize) {super(maxSize,Sugar.class);}
  public CompletableFuture<Sugar> make() {
    return CompletableFuture.supplyAsync(()->new Sugar());}
  }

class Charles extends Producer<Cake>{
  ActorRef alice; ActorRef bob;
  public Charles(ActorRef alice,ActorRef bob,int maxSize) {
    super(maxSize,Cake.class); this.alice=alice; this.bob=bob;}

  public CompletableFuture<Cake> make() {//equivalent
    CompletableFuture<Object> wheat=Patterns.ask(alice, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    CompletableFuture<Object> sugar=Patterns.ask(bob, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    return wheat.thenCombineAsync(sugar,(w,s)->new Cake(s, w)));
    //or even wheat.thenCombineAsync(sugar,Cake::new);
    }}
```

Our Producers

```
//variations


return CompletableFuture.supplyAsync(()->new Wheat());
  //1: simple, ok for possibly long but crazy long/ blocking computation

return CompletableFuture.completedFuture(new Wheat());
  //2: optimized, good for very short operations.

return CompletableFuture.supplyAsync(()->new Wheat(),myThreadPool);}
  //3: when the compuitation can be VERY long or blocking, is better to use your
own thread pool (like an Executors.newCachedThreadPool() ), to not exhaust the
default one.

//THREAD POOL EXAUSTION IS NOT A JAVA SPECIFIC PROBLEM :-)
```

Running producers

- In a producer,
  - running=true iff there are MakeOne messages around

- Those kinds of logic invariants are sprinkled all over programs. One of the keys to write correct programs is to
  - notice them
  - test them
  - assert them (assert keyword) when possible

Tim, the consumer

```
class Tim extends AbstractActor{
  int hunger;
  ActorRef charles;
  boolean running=true;
  ActorRef originalSender=null;
  Tim(int hunger,ActorRef charles){this.hunger=hunger; this.charles=charles;}

  private void askCakeOrGiveGift(){
    if(hunger<=0) {
      running=false;
      originalSender.tell(new Gift(),self());
      return;
      }
    CompletableFuture<Object> cake=Patterns.ask(charles, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    Patterns.pipe(cake, context().dispatcher()).to(self());
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiftRequest.class,()->originalSender==null,gr->{
        originalSender=sender(); askCakeOrGiveGift();})
      .match(Cake.class,()->running,c->{
        hunger-=1; askCakeOrGiveGift();})
      .build();}}
```

Tim, the consumer

```
class Tim extends AbstractActor{
  int hunger;
  ActorRef charles;
  boolean running=true;
  ActorRef originalSender=null;
  Tim(int hunger,ActorRef charles){this.hunger=hunger; this.charles=charles;}

  private void askCakeOrGiveGift(){
    if(hunger<=0) {
      running=false;
      originalSender.tell(new Gift(),self());
      return;
      }
    CompletableFuture<Object> cake=Patterns.ask(charles, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    Patterns.pipe(cake, context().dispatcher()).to(self());
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiftRequest.class,()->originalSender==null,gr->{
        originalSender=sender(); askCakeOrGiveGift();})
      .match(Cake.class,()->running,c->{
        hunger-=1; askCakeOrGiveGift();})
      .build();}}
```

Tim, the consumer

```
class Tim extends AbstractActor{
  int hunger;
  ActorRef charles;
  boolean running=true;
  ActorRef originalSender=null;
  Tim(int hunger,ActorRef charles){this.hunger=hunger; this.charles=charles;}

  private void askCakeOrGiveGift(){
    if(hunger<=0) {
      running=false;
      originalSender.tell(new Gift(),self());
      return;
      }
    CompletableFuture<Object> cake=Patterns.ask(charles, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    Patterns.pipe(cake, context().dispatcher()).to(self());
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiftRequest.class,()->originalSender==null,gr->{
        originalSender=sender(); askCakeOrGiveGift();})
      .match(Cake.class,()->running,c->{
        hunger-=1; askCakeOrGiveGift();})
      .build();}}
```

Tim, the consumer

```
class Tim extends AbstractActor{
  int hunger;
  ActorRef charles;
  boolean running=true;
  ActorRef originalSender=null;
  Tim(int hunger,ActorRef charles){this.hunger=hunger; this.charles=charles;}

  private void askCakeOrGiveGift(){
    if(hunger<=0) {
      running=false;
      originalSender.tell(new Gift(),self());
      return;
      }
    CompletableFuture<Object> cake=Patterns.ask(charles, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    Patterns.pipe(cake, context().dispatcher()).to(self());
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiftRequest.class,()->originalSender==null,gr->{
        originalSender=sender(); askCakeOrGiveGift();})
      .match(Cake.class,()->running,c->{
        hunger-=1; askCakeOrGiveGift();})
      .build();}}
```

Tim, the consumer

Correct but unsatisfactory! (see later)

```java
class Tim extends AbstractActor{
  int hunger;
  ActorRef charles;
  boolean running=true;
  ActorRef originalSender=null;
  Tim(int hunger,ActorRef charles){this.hunger=hunger; this.charles=charles;}

  private void askCakeOrGiveGift(){
    if(hunger<=0) {
      running=false;
      originalSender.tell(new Gift(),self());
      return;
      }
    CompletableFuture<Object> cake=Patterns.ask(charles, GiveOne.instance,
      Duration.ofMillis(5000)).toCompletableFuture();
    Patterns.pipe(cake, context().dispatcher()).to(self());
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(GiftRequest.class,()->originalSender==null,gr->{
        originalSender=sender(); askCakeOrGiveGift();})
      .match(Cake.class,()->running,c->{
        hunger-=1; askCakeOrGiveGift();})
      .build();}}
```

Putting all together

```
public static Gift computeGift(int hunger){
  ActorSystem s=AkkaConfig.newSystem("Cakes", 2501, /*..*/);

  ActorRef alice=//makes wheat
    s.actorOf(Props.create(Alice.class,()->new Alice(100)),"Alice");

  ActorRef bob=//makes sugar
    s.actorOf(Props.create(Bob.class,()->new Bob(100)),"Bob");

  ActorRef charles=// makes cakes with wheat and sugar
    s.actorOf(Props.create(Charles.class,
      ()->new Charles(alice,bob,100)),"Charles");

  ActorRef tim=//tim wants to eat cakes
    s.actorOf(Props.create(Tim.class,()->new Tim(hunger,charles)),"Tim");

  CompletableFuture<Object> gift = Patterns.ask(tim,new GiftRequest(),
    Duration.ofMinutes(10000)).toCompletableFuture();
  try{ return (Gift)gift.join(); }
  finally{
    alice.tell(PoisonPill.getInstance(),ActorRef.noSender());
    bob.tell(PoisonPill.getInstance(),ActorRef.noSender());
    charles.tell(PoisonPill.getInstance(),ActorRef.noSender());
    tim.tell(PoisonPill.getInstance(),ActorRef.noSender());
    s.terminate();}
  }
```

Putting all together

```
// Non serializable Lambda

    ActorRef charles=// makes cakes with wheat and sugar
      s.actorOf(Props.create(Charles.class,
          (akka.japi.Creator<Charles> &Serializable)
          ()->new Charles(alice,bob,100)),"Charles");
```

- Lambdas are serializable if the interface they implements extends Serializable.
- You can force serializable lambdas with the syntax above. No more needed; it was needed in the past. It may be depending on the specific version of Java/Akka
- A lambda can be serialized if:
  - the lambda type is serializable
  - all the objects captured by the lambda can be serialized.
- No serialization = no distribution on multiple machines.

Can we do better?

- Why do I need to make a new class for any new producer?
- Tim is not a producer, what is Tim? can we abstract over the of actors like him?
-
- Actor and ActorRef: to provide new abstractions with a good API we need to take care of both Actors and ActorRefs, and wrap both in some way.
- Do we still remember the ConcurrentRunner<T>?

Get<T>: a generic producer

```
public interface Get<T> extends Serializable{

  CompletableFuture<T> get();

  default void stop() {}

  static final class LambdaProducer<T> extends Producer<T>{
    private Get<T> maker;
    private LambdaProducer(int maxSize, Class<T> product, Get<T> maker) {
      super(maxSize, product); this.maker=maker;}

    public CompletableFuture<T> make() {return maker.get();}
    }

  @SuppressWarnings({ "unchecked", "serial" })
  public static <P> Get<P> producer(ActorSystem system, String name,
      int maxSize,Class<P> product, Get<P> maker){
    ActorRef ref=system.actorOf(Props.create(Get.LambdaProducer.class,
      ()->new Get.LambdaProducer<P>(maxSize,product,maker)),name);

    return new Get<P>(){
      public CompletableFuture<P> get(){
        return(CompletableFuture<P>)Patterns.ask(ref,GiveOne.instance,
          Duration.ofMinutes(5000)).toCompletableFuture();
        }
      public void stop() {ref.tell(PoisonPill.getInstance(),ActorRef.noSender());}
      };
    } }
```

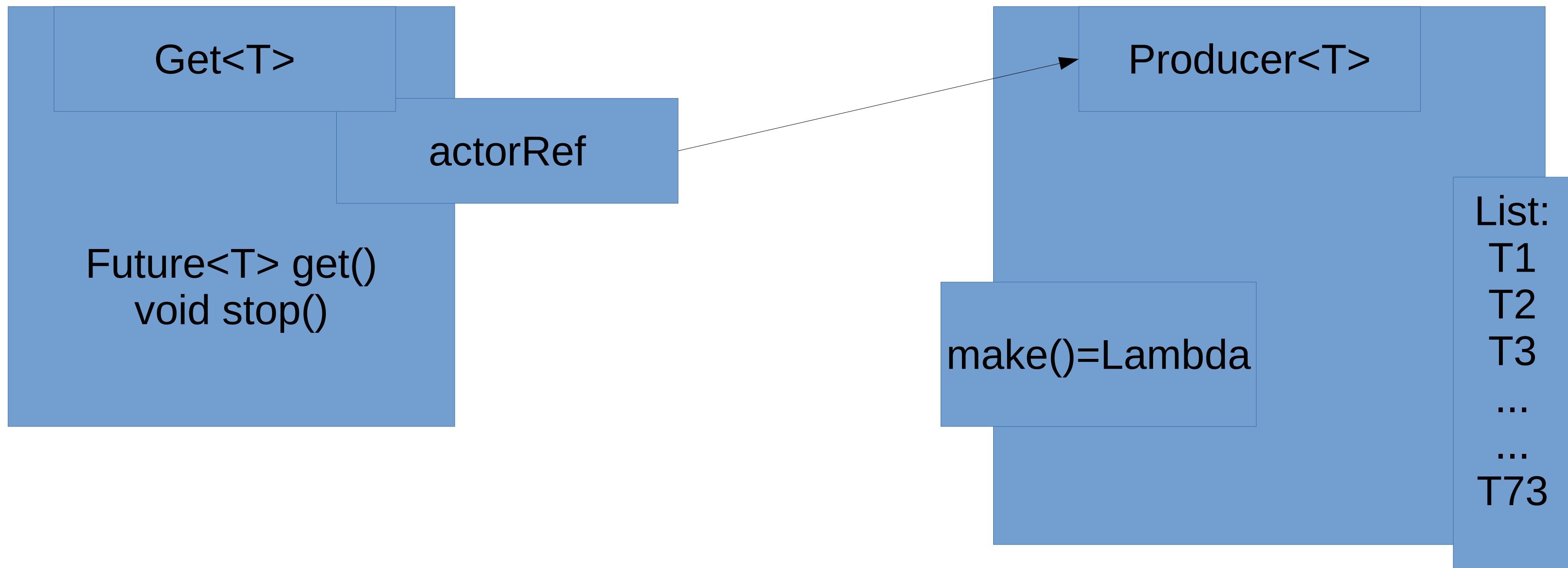
Get<T>: a generic producer

```
public interface Get<T> extends Serializable{

  CompletableFuture<T> get();

  default void stop() {}

  static final class LambdaProducer<T> extends Producer<T>{
    private Get<T> maker;
    private LambdaProducer(int maxSize, Class<T> product, Get<T> maker) {
      super(maxSize, product); this.maker=maker;}

    public CompletableFuture<T> make() {return maker.get();}
    }

  @SuppressWarnings({ "unchecked", "serial" })
  public static <P> Get<P> producer(ActorSystem system, String name,
      int maxSize,Class<P> product, Get<P> maker){
    ActorRef ref=system.actorOf(Props.create(Get.LambdaProducer.class,
      ()->new Get.LambdaProducer<P>(maxSize,product,maker)),name);

    return new Get<P>(){
      public CompletableFuture<P> get(){
        return(CompletableFuture<P>)Patterns.ask(ref,GiveOne.instance,
          Duration.ofMinutes(5000)).toCompletableFuture();
        }
      public void stop() {ref.tell(PoisonPill.getInstance(),ActorRef.noSender());}
      };
    } }
```

Get<T>: a generic producer

```
public interface Get<T> extends Serializable{

  CompletableFuture<T> get();

  default void stop() {}

  static final class LambdaProducer<T> extends Producer<T>{
    private Get<T> maker;
    private LambdaProducer(int maxSize, Class<T> product, Get<T> maker) {
      super(maxSize, product); this.maker=maker;}

    public CompletableFuture<T> make() {return maker.get();}
    }

  @SuppressWarnings({ "unchecked", "serial" })
  public static <P> Get<P> producer(ActorSystem system, String name,
      int maxSize,Class<P> product, Get<P> maker){
    ActorRef ref=system.actorOf(Props.create(Get.LambdaProducer.class,
      ()->new Get.LambdaProducer<P>(maxSize,product,maker)),name);

    return new Get<P>(){
      public CompletableFuture<P> get(){
        return(CompletableFuture<P>)Patterns.ask(ref,GiveOne.instance,
          Duration.ofMinutes(5000)).toCompletableFuture();
        }
      public void stop() {ref.tell(PoisonPill.getInstance(),ActorRef.noSender());}
      };
    } }
```

Get<T>



- A Get object is deeply immutable, since it only contains the deeply immutable actor ref. Thus actors can share Get freely

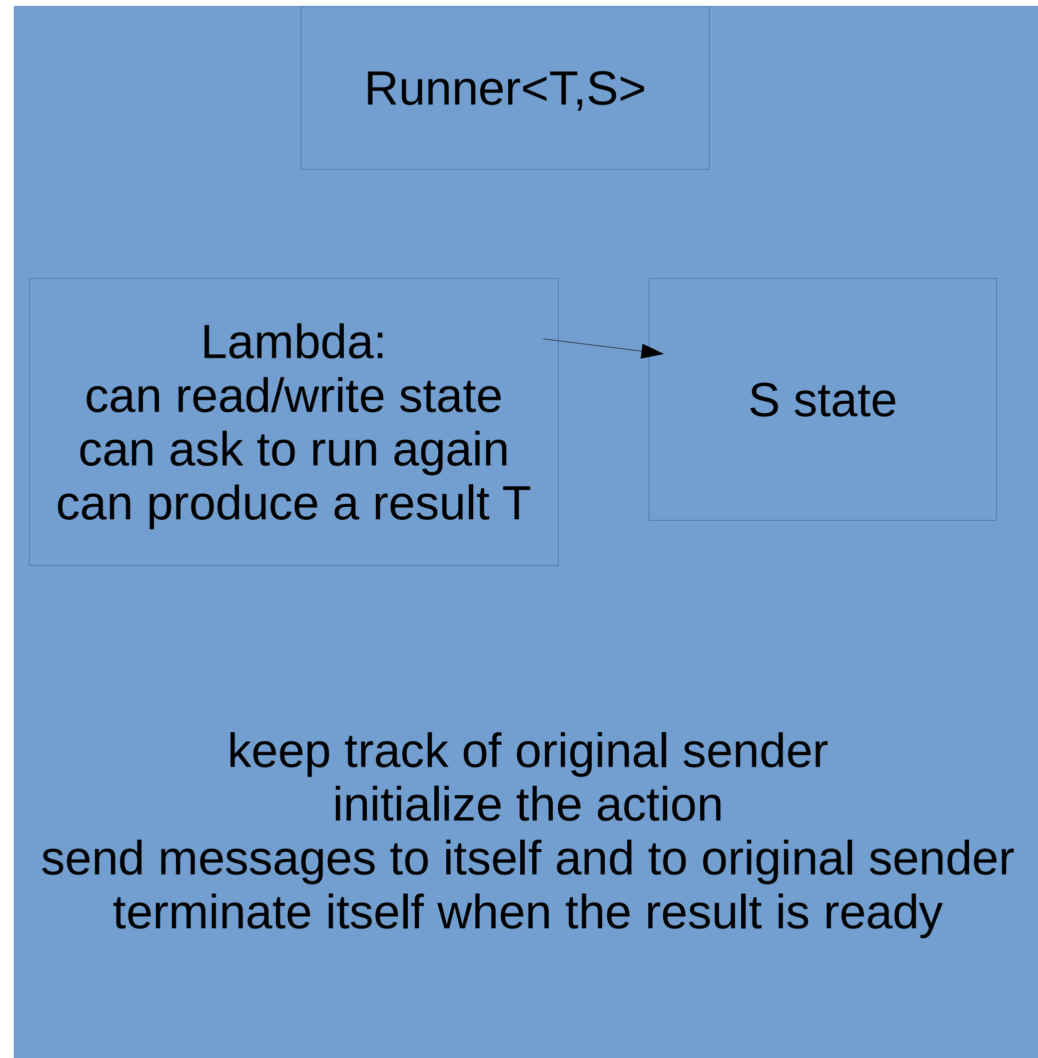
Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{
  private Consumer<Runner<T,S>> r;
  private ActorRef originalSender;
  volatile private S state;
  private Runner(Consumer<Runner<T,S>> r, S state){this.r=r; this.state=state;}
  public S getState(){return state;}
  public void setState(S state){this.state=state;}

  public void setResult(T res) {originalSender.tell(res,self());}
  public void repeat() {self().tell(GiveOne.instance,self());}

  public Receive createReceive(){
    return receiveBuilder().match(GiveOne.class,this::act).build();
    }
  private void act(GiveOne g){
    if(this.originalSender==null){originalSender=sender();}
    r.accept(this);
    }
  public static <S,T>CompletableFuture<T> run(ActorSystem system, String name,
      S initialState, Consumer<Runner<T,S>> r){
    ActorRef ref=system.actorOf(Props.create(Runner.class,
      ()->new Runner<T,S>(r,initialState)),name);
    @SuppressWarnings("unchecked")
    CompletableFuture<T> res=(CompletableFuture<T>)Patterns.ask(ref,
      GiveOne.instance,Duration.ofMinutes(10000)).toCompletableFuture();
    res.thenAccept(unused->ref.tell(PoisonPill.getInstance(),ActorRef.noSender()));
    return res;
    }  }
```

Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{// T=return type, S=internal state
  private Consumer<Runner<T,S>> r; //our personalized behavior.
  private ActorRef originalSender;
  volatile private S state;//volatile: it can be accessed by many workers
  private Runner(Consumer<Runner<T,S>> r, S state){this.r=r; this.state=state;}
  public S getState(){return state;} //our personalized behavior can get state
  public void setState(S state){this.state=state;} //and set state

  public void setResult(T res) {originalSender.tell(res,self());}//and return a T
  public void repeat() {self().tell(GiveOne.instance,self());}//and run again!

  public Receive createReceive(){
    return receiveBuilder().match(GiveOne.class,this::act).build();
    }
  private void act(GiveOne g){
    if(this.originalSender==null){originalSender=sender();}
    r.accept(this);
    }
  public static <S,T>CompletableFuture<T> run(ActorSystem system, String name,
      S initialState, Consumer<Runner<T,S>> r){
    ActorRef ref=system.actorOf(Props.create(Runner.class,
      ()->new Runner<T,S>(r,initialState)),name);
    @SuppressWarnings("unchecked")
    CompletableFuture<T> res=(CompletableFuture<T>)Patterns.ask(ref,
      GiveOne.instance,Duration.ofMinutes(10000)).toCompletableFuture();
    res.thenAccept(unused->ref.tell(PoisonPill.getInstance(),ActorRef.noSender()));
    return res;
    }  }
```

Runner: a wrapper actor



Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{
  private Consumer<Runner<T,S>> r;
  private ActorRef originalSender;
  volatile private S state;
  private Runner(Consumer<Runner<T,S>> r, S state){this.r=r; this.state=state;}
  public S getState(){return state;}
  public void setState(S state){this.state=state;}

  public void setResult(T res) {originalSender.tell(res,self());}
  public void repeat() {self().tell(GiveOne.instance,self());}

  public Receive createReceive(){
    return receiveBuilder().match(GiveOne.class,this::act).build();
    }
  private void act(GiveOne g){
    if(this.originalSender==null){originalSender=sender();}
    r.accept(this);
    }
  public static <S,T>CompletableFuture<T> run(ActorSystem system, String name,
      S initialState, Consumer<Runner<T,S>> r){
    ActorRef ref=system.actorOf(Props.create(Runner.class,
      ()->new Runner<T,S>(r,initialState)),name);
    @SuppressWarnings("unchecked")
    CompletableFuture<T> res=(CompletableFuture<T>)Patterns.ask(ref,
      GiveOne.instance,Duration.ofMinutes(10000)).toCompletableFuture();
    res.thenAccept(unused->ref.tell(PoisonPill.getInstance(),ActorRef.noSender()));
    return res;
    }  }
```

Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{
  private Consumer<Runner<T,S>> r;
  private ActorRef originalSender;
  volatile private S state;
  private Runner(Consumer<Runner<T,S>> r, S state){this.r=r; this.state=state;}
  public S getState(){return state;}
  public void setState(S state){this.state=state;}

  public void setResult(T res) {originalSender.tell(res,self());}
  public void repeat() {self().tell(GiveOne.instance,self());}

  public Receive createReceive(){
    return receiveBuilder().match(GiveOne.class,this::act).build();
    }
  private void act(GiveOne g){
    if(this.originalSender==null){originalSender=sender();}
    r.accept(this);
    }
  public static <S,T>CompletableFuture<T> run(ActorSystem system, String name,
      S initialState, Consumer<Runner<T,S>> r){//set up itself:
    ActorRef ref=system.actorOf(Props.create(Runner.class,//ref to Runner
      ()->new Runner<T,S>(r,initialState)),name);
    @SuppressWarnings("unchecked") //the ask pattern for final result
    CompletableFuture<T> res=(CompletableFuture<T>)Patterns.ask(ref,
      GiveOne.instance,Duration.ofMinutes(10000)).toCompletableFuture();
    res.thenAccept(unused->ref.tell(PoisonPill.getInstance(),ActorRef.noSender()));
    return res; //set up the Runner to die after the result is completed.
    }  }
```

Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{
  private Consumer<Runner<T,S>> r;
  private ActorRef originalSender;
  volatile private S state;
  private Runner(Consumer<Runner<T,S>> r, S state){this.r=r; this.state=state;}
  public S getState(){return state;}
  public void setState(S state){this.state=state;}

  public void setResult(T res) {originalSender.tell(res,self());}
  public void repeat() {self().tell(GiveOne.instance,self());}

  public Receive createReceive(){
    return receiveBuilder().match(GiveOne.class,this::act).build();
    }
  private void act(GiveOne g){
    if(this.originalSender==null){originalSender=sender();}
    r.accept(this);
    }
  public static <S,T>CompletableFuture<T> run(ActorSystem system, String name,
      S initialState, Consumer<Runner<T,S>> r){
    ActorRef ref=system.actorOf(Props.create(Runner.class,
      ()->new Runner<T,S>(r,initialState)),name);
    @SuppressWarnings("unchecked")
    CompletableFuture<T> res=(CompletableFuture<T>)Patterns.ask(ref,
      GiveOne.instance,Duration.ofMinutes(10000)).toCompletableFuture();
    res.thenAccept(unused->ref.tell(PoisonPill.getInstance(),ActorRef.noSender()));
    return res;
    }  }
```

Runner: a wrapper actor

```
final class Runner<T,S> extends AbstractActor{//better code:
  ...
  volatile boolean repeating=false;//true iff repeat() has been called already

  public void repeat() {//constraint usage: set repeating only once
    if(this.originalSender==null) {throw new Error("result already set");}
    if(this.repeating) {throw new Error("repeating already set");}
    this.repeating=true;
    self().tell(GiveOne.instance,self());
    }

  public void setResult(T res) {//constraint usage: set result only once
    if(this.originalSender==null) {throw new Error("result already set");}
    this.originalSender.tell(res,self());
    this.originalSender=null;
    }
  ...
  }
```

Usage

```
ActorSystem system = AkkaConfig.newSystem("cakes",2501,Map.of());

Get<Wheat> alice = Get.producer(system, "Alice", 100, Wheat.class,
  ()->CompletableFuture.supplyAsync(Wheat::new));

Get<Sugar> bob = Get.producer(system, "Bob", 100, Sugar.class,
  ()->CompletableFuture.supplyAsync(Sugar::new));

Get<Cake> charles = Get.producer(system, "Charles", 100, Cake.class,
  ()->bob.get().thenCombineAsync(alice.get(),Cake::new));

CompletableFuture<Gift> res = Runner.run(system, "Tim", hunger,
  r->charles.get().thenAcceptAsync(cake->{//r is the Runner
    if(r.getState()<=0){  //r have getState Integer
      r.setResult(new Gift());return; //and setResult
      }
    r.setState(r.getState()-1);
    r.repeat(); //repeat to run again
    })
  );
try{return res.join();}
finally {alice.stop();bob.stop();charles.stop();system.terminate();}
```

Comparison

```
//Our old ConcurrentRunner from lecture 11, pure Java standard library
ConcurrentRunner<Gift> runner=new ConcurrentRunner<>();

BlockingQueue<Wheat> ws=new LinkedBlockingQueue<>(100);
BlockingQueue<Sugar> ss=new LinkedBlockingQueue<>(100);
BlockingQueue<Cake>  cs=new LinkedBlockingQueue<>(100);
AtomicInteger timHunger=new AtomicInteger(hunger);


runner.run(()->ws.put(new Wheat()));//Alice

runner.run(()->ss.put(new Ssugar()));//Bob

runner.run(()->cs.put(new Cake(ss.take(),ws.take())));//Charles

runner.run(()->{//Tim
  if(timHunger.decrementAndGet()>0){return;}
  runner.setResult(new Gift());
  });

return runner.result();
```

Comparing performance

- Similarly, as for task 3, I made a busy loop to delay the creation of Sugar, Wheat and Cake; same delay for all of them in this case.
- Tested on a single machine with different level of delay.
- I expected the Akka version to be slower then the pure Java version, and that the only advantage of Akka would have been in either using many machines, or a much larger network of producer/consumers.
- ***I WAS SO WRONG***

Results

With delay=200	Akka: 1.426.625	Java8: 2.000.200
With delay=150	Akka: 1.105.823	Java8: 1.500.150
With delay=100	Akka: 733.223	Java8: 1.000.100
With delay=75	Akka: 504.664	Java8: 750.076
With delay=50	Akka: 338.217	Java8: 500.050
With delay=25	Akka: 175.020	Java8: 250.025
With delay=15	Akka: 102.778	Java8: 150.016
With delay=10	Akka: 73.221	Java8: 100.010
With delay=7	Akka: 48.258	Java8: 70.007
With delay=5	Akka: 33.711	Java8: 50.005
With delay=4	Akka: 28.115	Java8: 40.004
With delay=3	Akka: 22.167	Java8: 30.003
With delay=2	Akka: 14.944	Java8: 20.002
With delay=1	Akka: 6.698	Java8: 10.001
With delay=0	Akka: 133	Java8: 47

Except when there is no delay, Akka lock-less algorithms beats Java8 locking ones.

That is the end of all course content

- In short, we learned:
  - avoid parallelism at first, but code so that you can add it later if need arise
  - using Streams is the best option when possible
  - using Futures/CompletableFutures otherwise
  - Actors: ideal for multiple machines, decent on single machine.
- Next lecture, we will review all the course material.
-
- Prepare questions and considerations.