I have seen the top of Akka Mountain, and it is good!

In One Sentence....

“To help you understand Actors, Supervision, Futures, Routers in both Scala and Java”

A note about my presentation style

- Trying to achieve the right mix of demonstration and slides.
- Code for this presentation is available on GitHub: `git@github.com:dhinojosa/akka-study.git`
- Goal is to provide you with code that you can use and reference with a presentation to back it.

About Akka

- Set of libraries used to create concurrent, fault-tolerant and scalable applications.
- It contains many API packages:
  - Actors, Logging, Futures, STM, Dispatchers, ...
- We are going to only focus on some of the core items.
- Managing processes in the same VM or in a Remote VM.

Game changer?

- Imagine life asynchronously
- Actors manage their own
- Web sites are faster because they delegated tasks
- What if you need something persisted later, don't necessarily need it right now?
- What if emails can be sent later on?

Actors

- Based on the Actor Model from Erlang.
- Encapsulate State and Behavior
- Concurrent processors that exchange messages.
- Each message is immutable (cannot be changed, this is required!)
- Each message should not be a closure
- Breath of fresh air if you have suffered concurrency

Actors (The basics)

- Create a message
- Send a message to an actor
- Each message has to be immutable
- Each message should not be a closure.

What does immutable look like? (Java)

```
public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() {...}
}
```

What does immutable look like? (Scala)

```
case class Person(val firstName:String, val lastName:String)
```

What does closure look like? (Java)

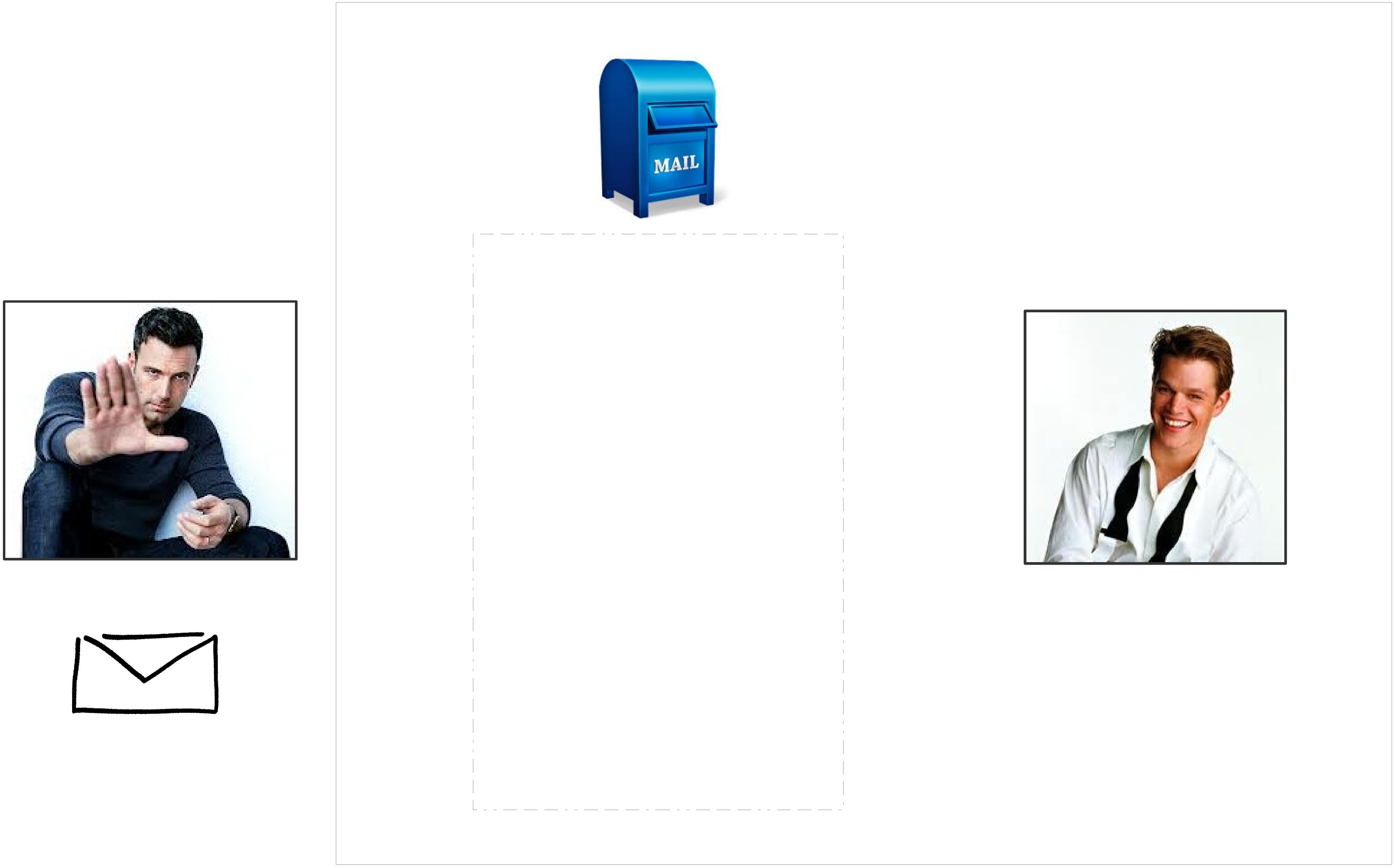
```
@see Java 8
```

What does closure look like? (Scala)

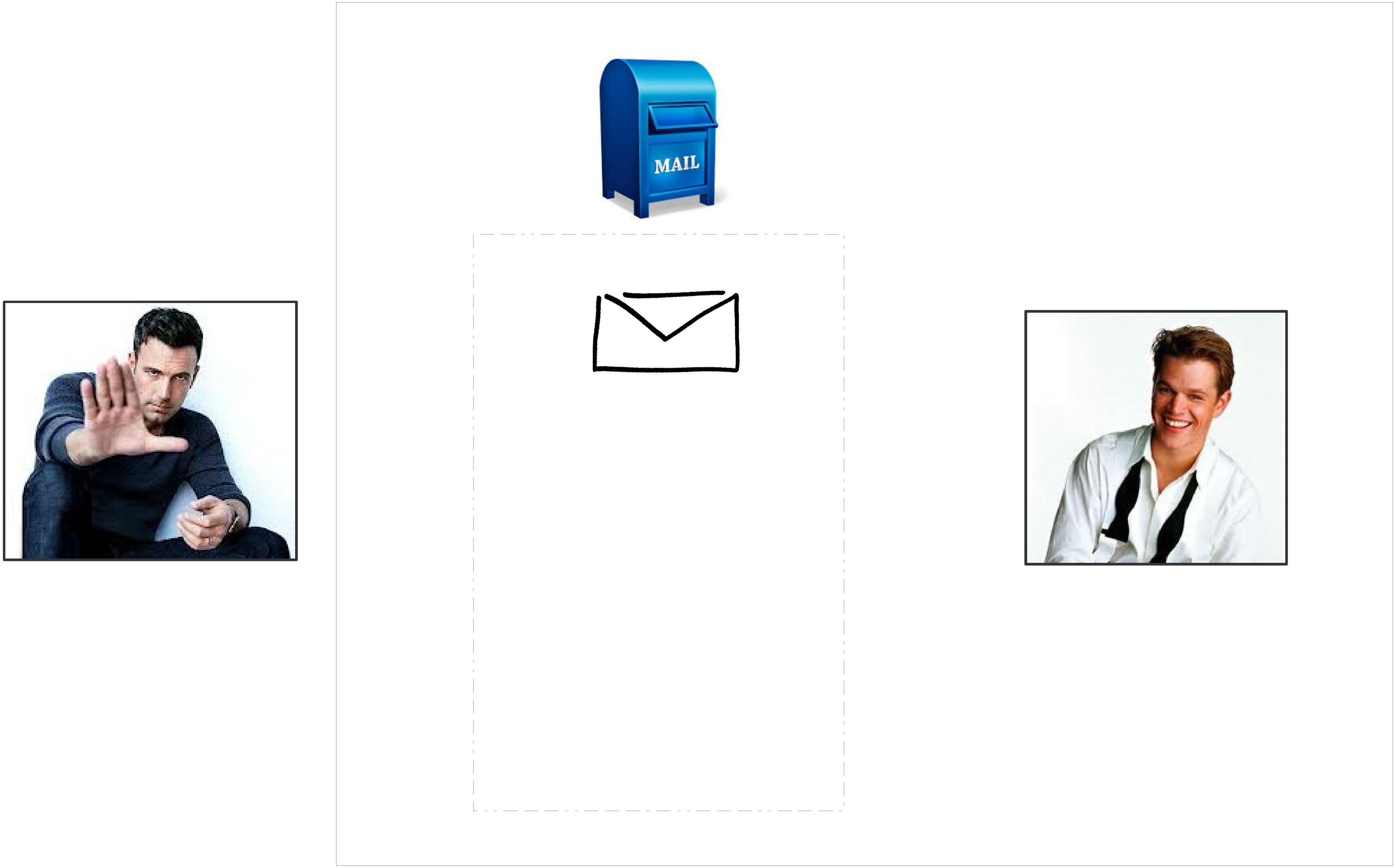
```
var x = 3

val y = {x:Int => x + 4}

def foo(w: Int => Int) = w(5)

foo(y) //9
```

Inside the Actor's Studio

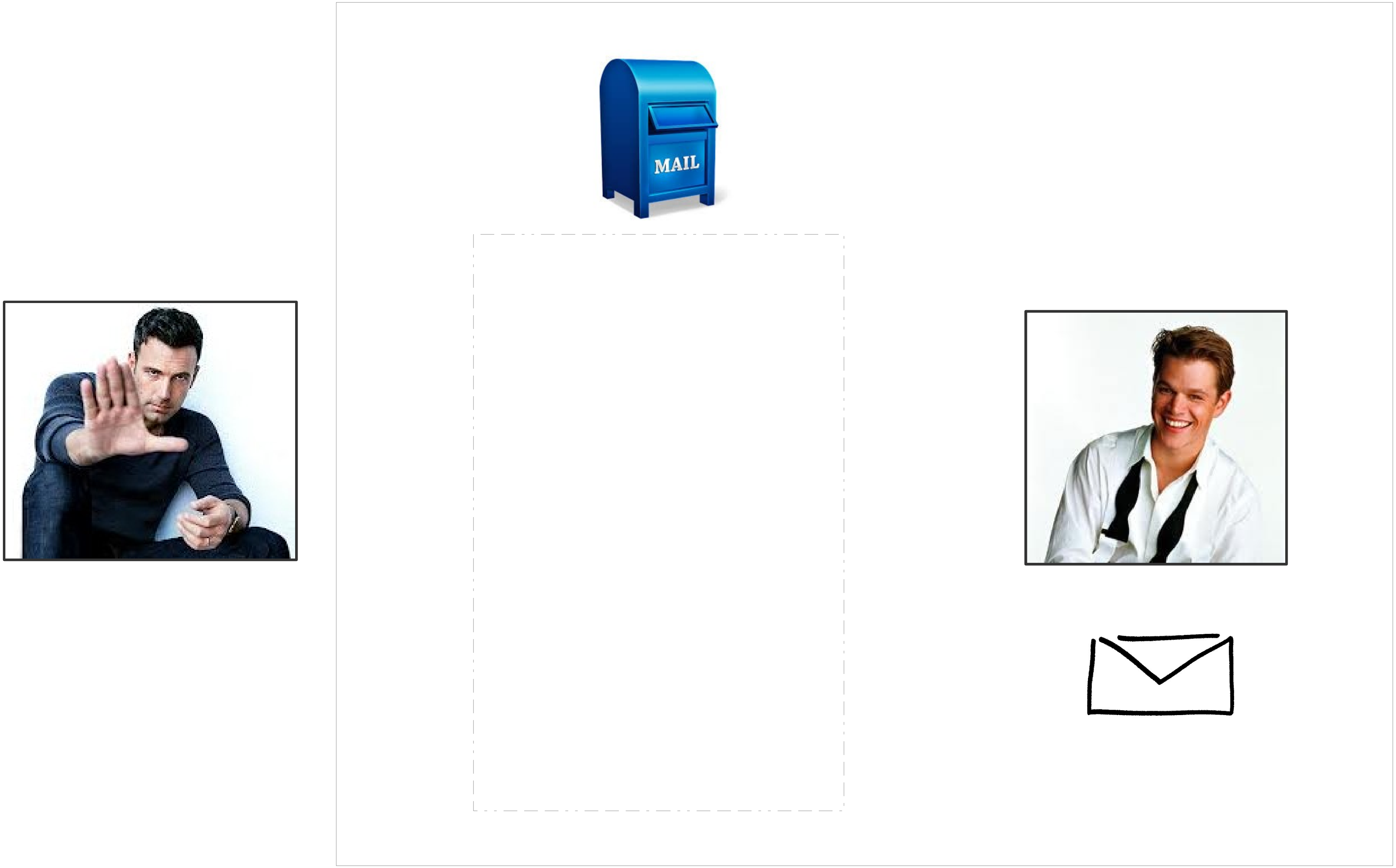
Actor Visual



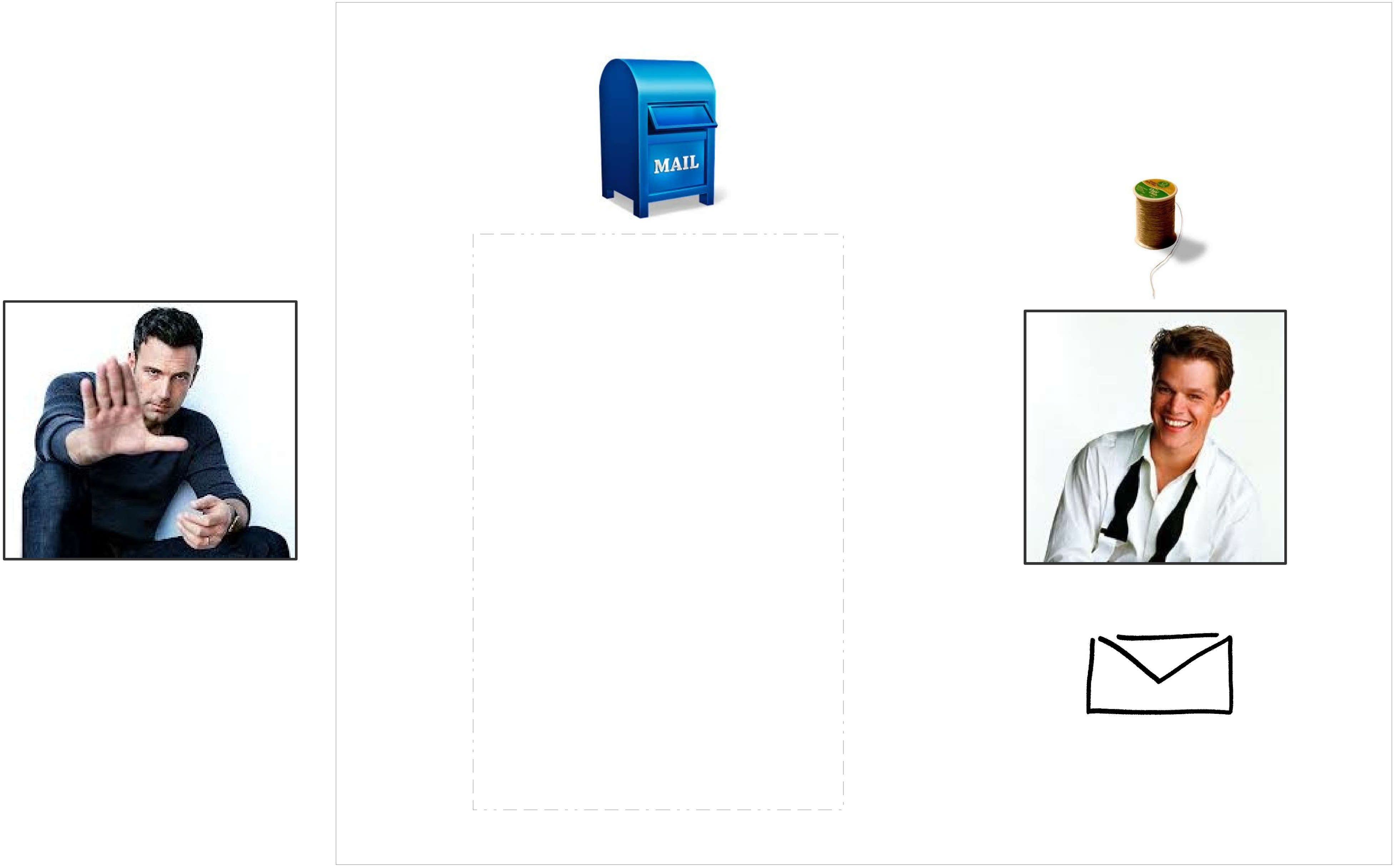
Actor Visual



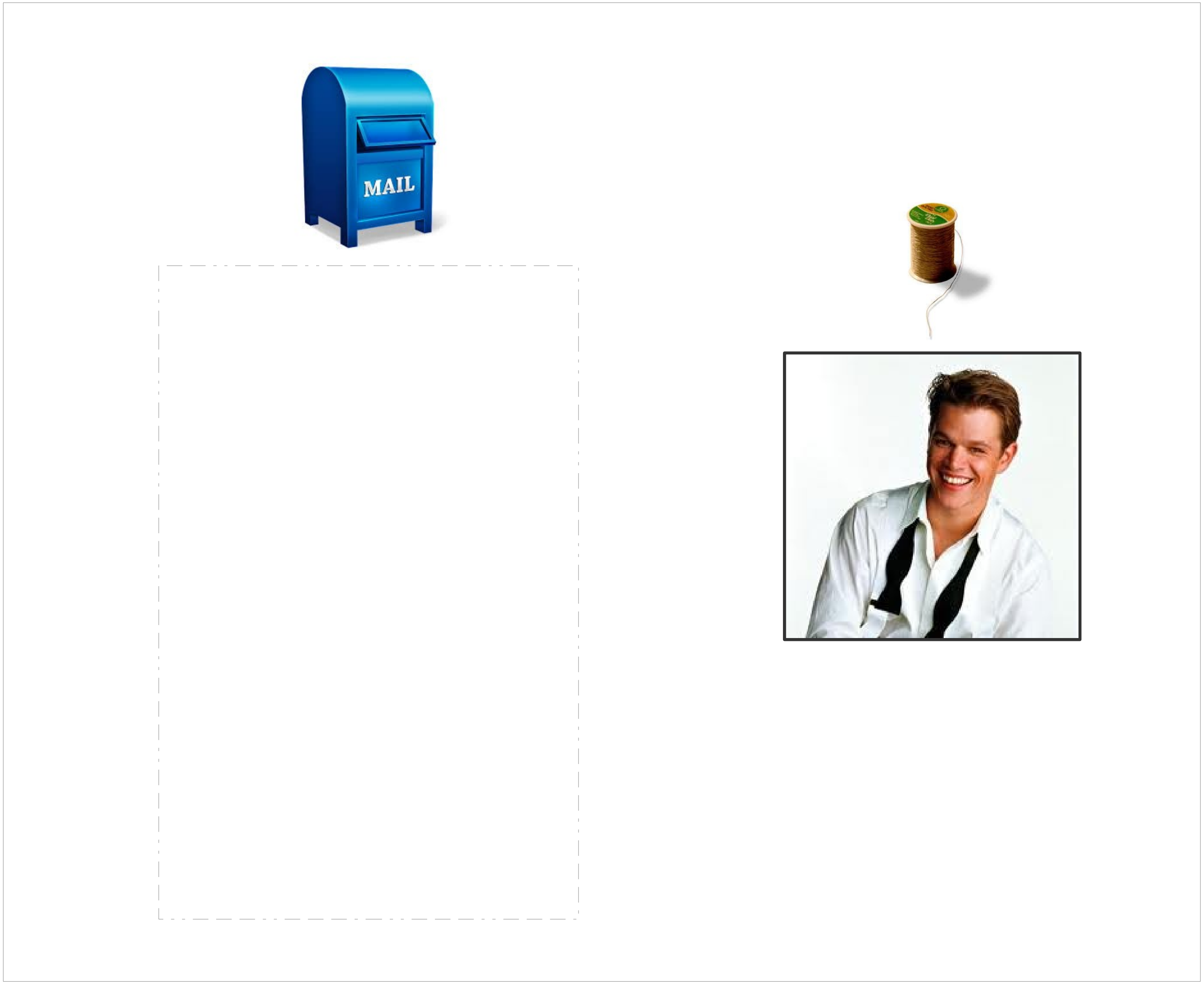
Actor Visual



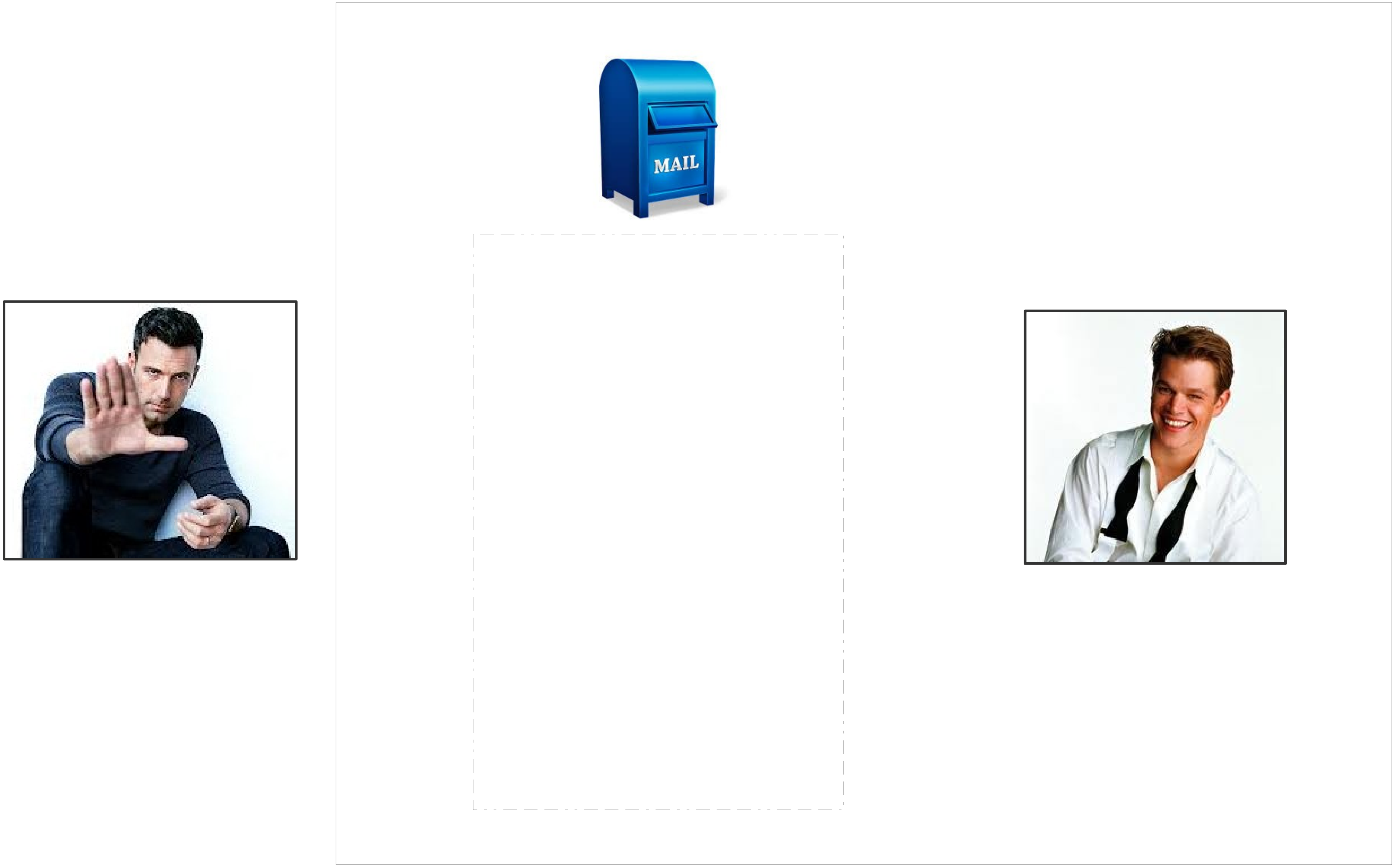
Actor Visual



Actor Visual

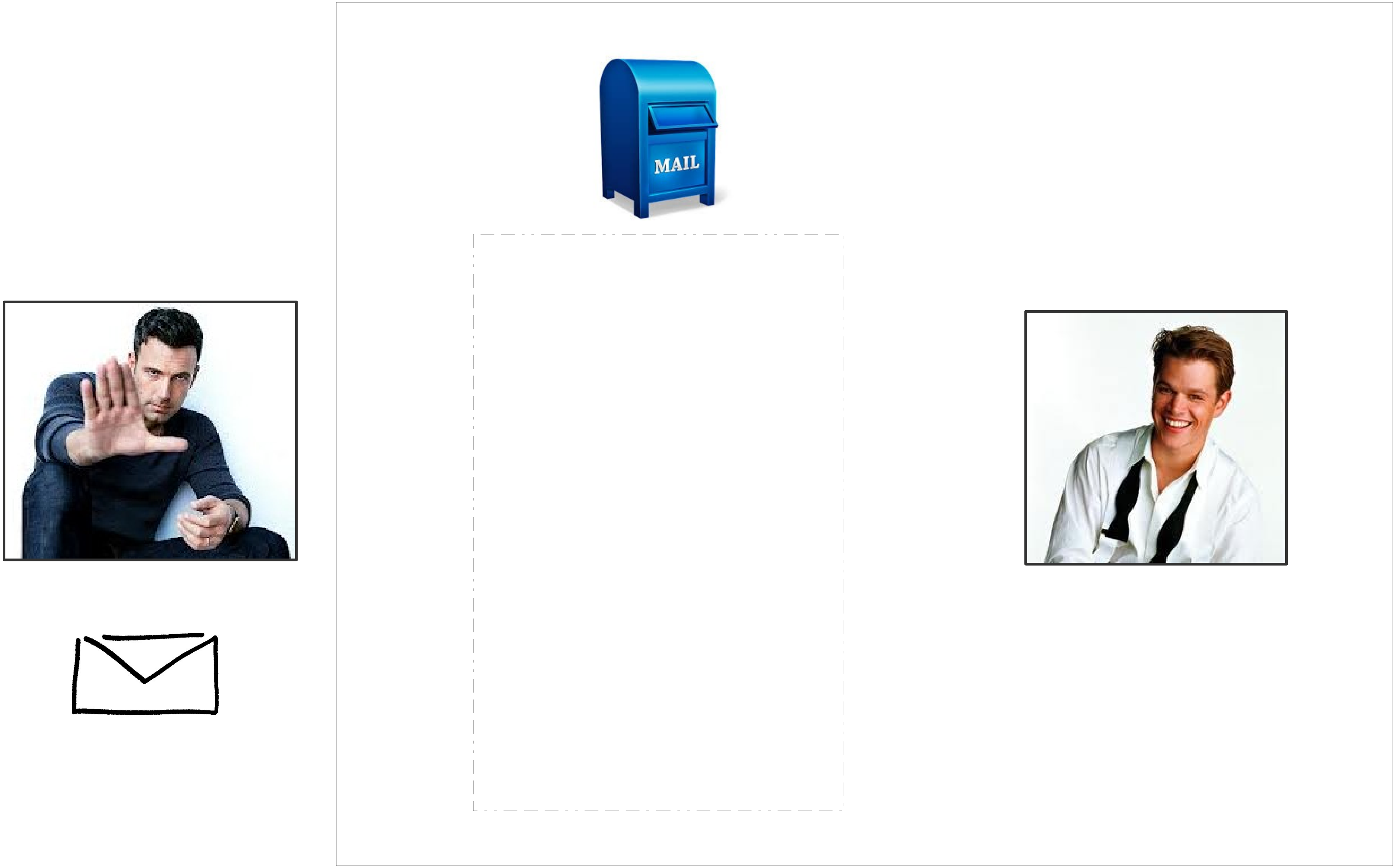


Actor Visual

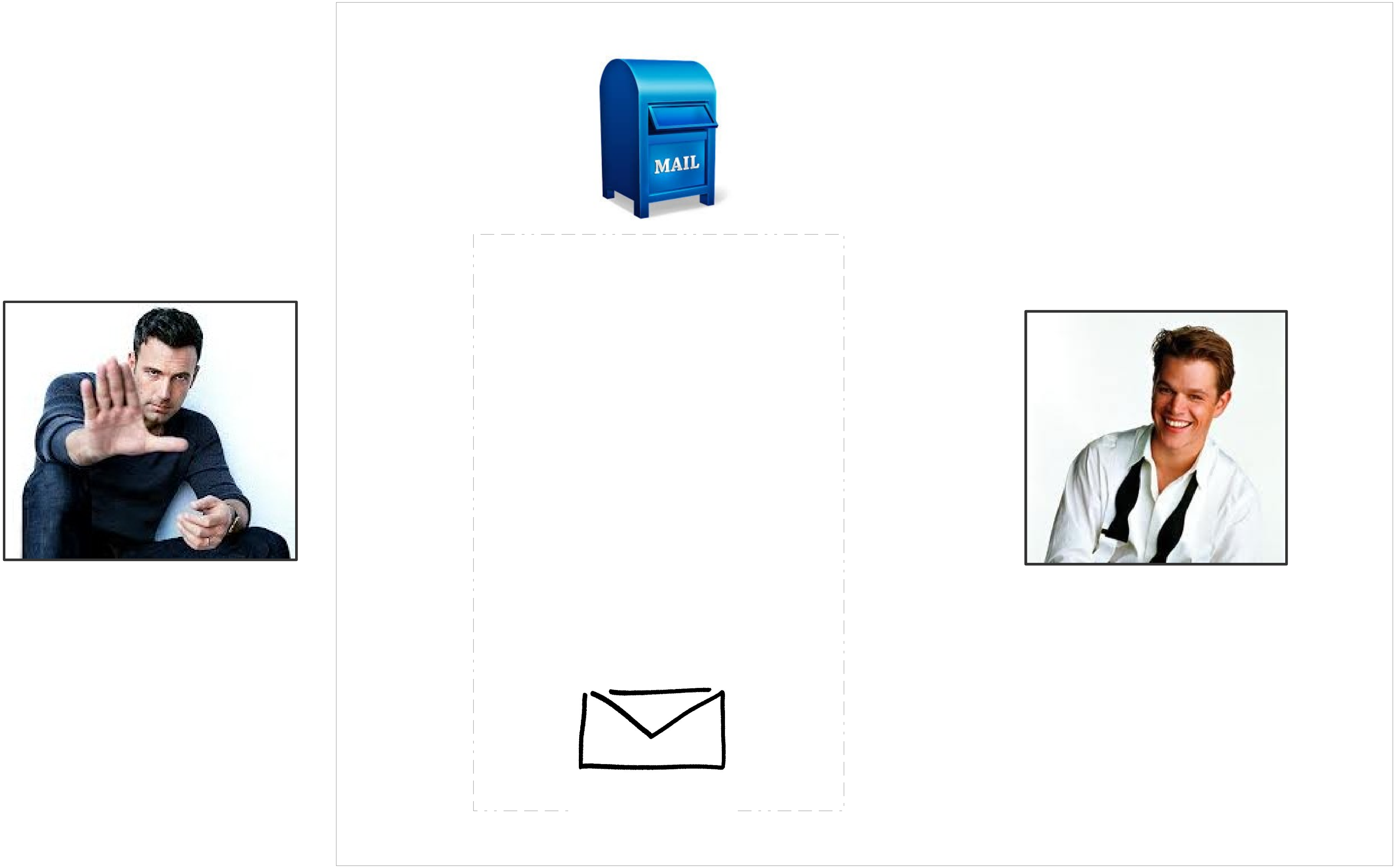


Inside the Actor's Studio Again

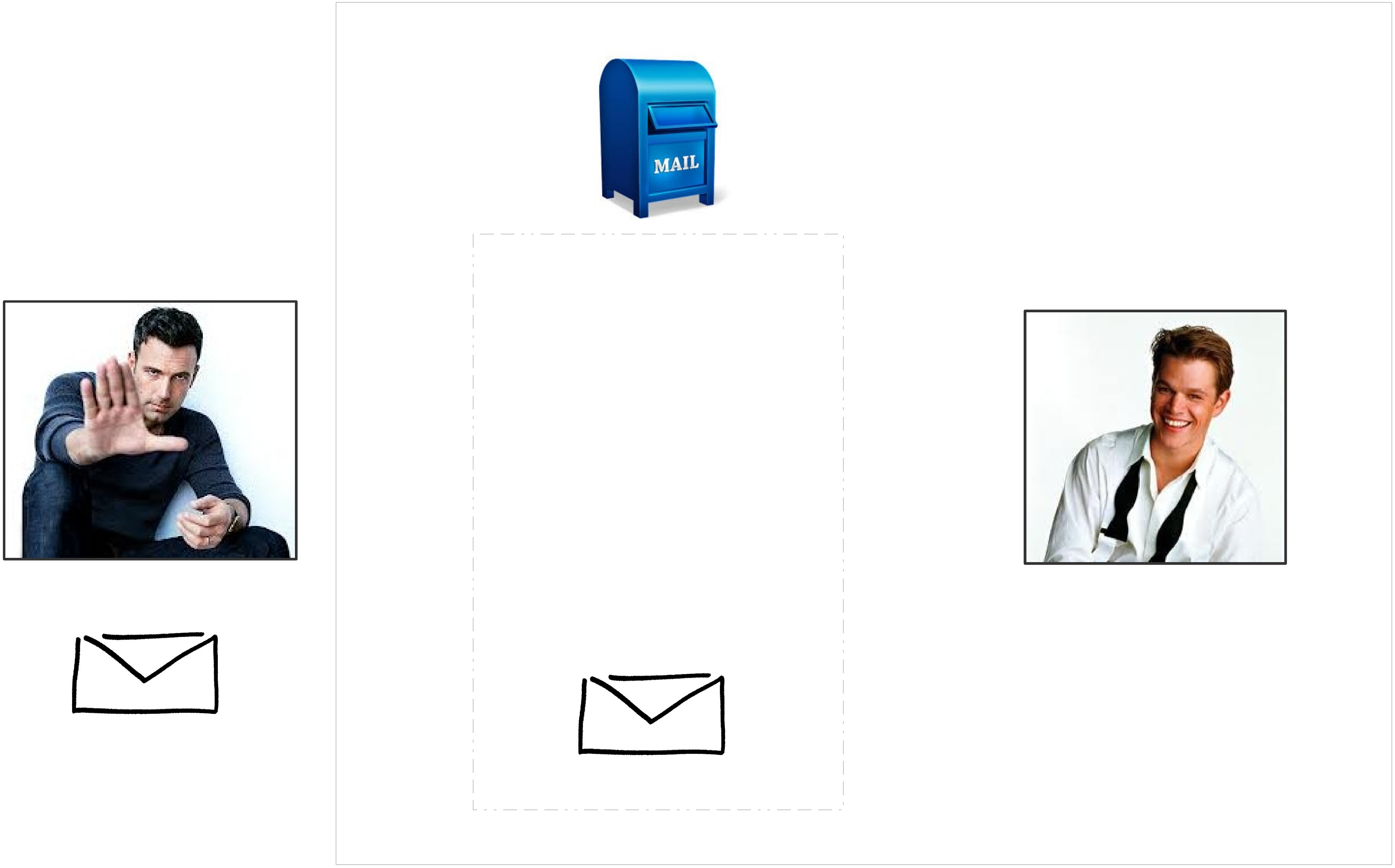
Actor Visual



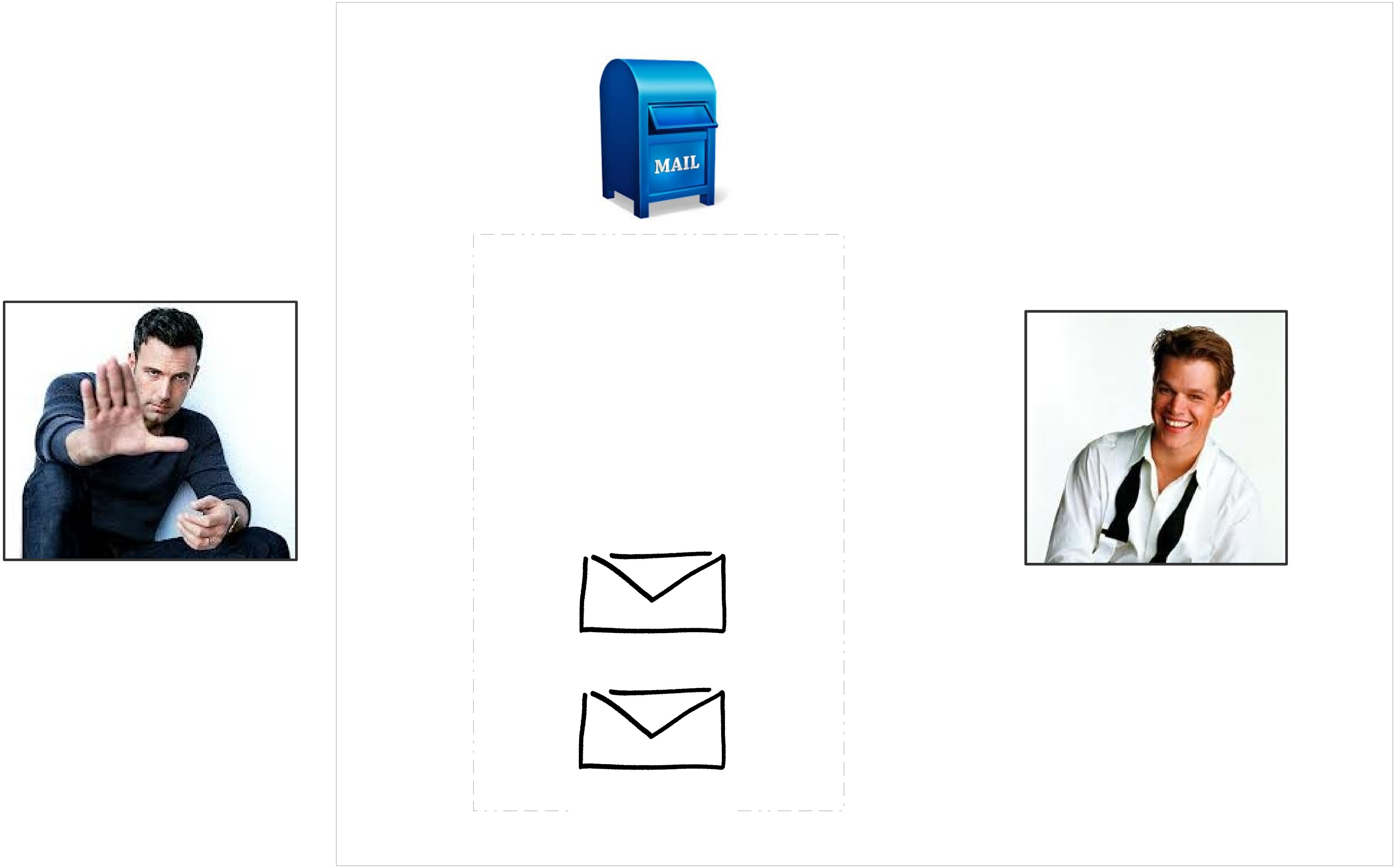
Actor Visual



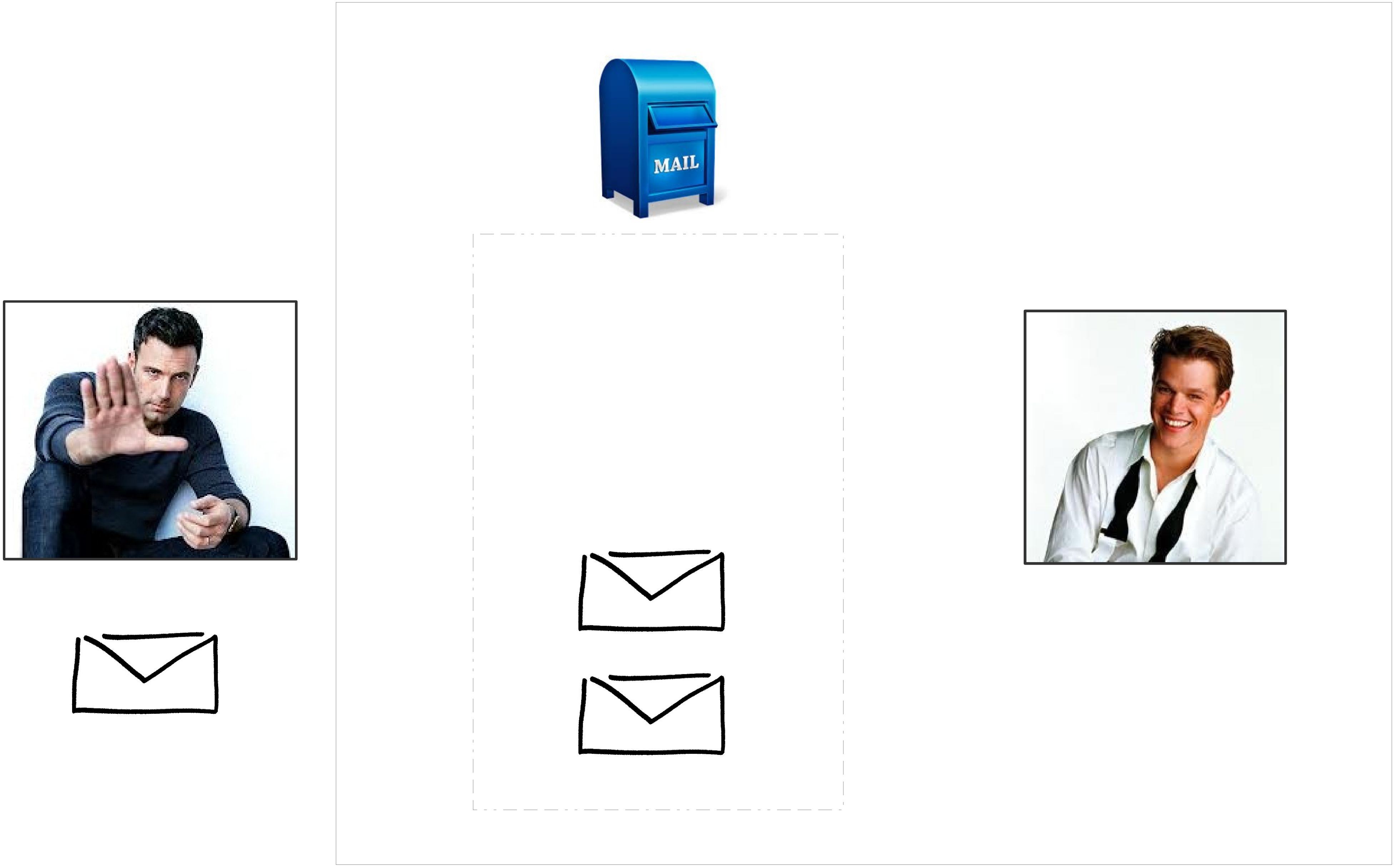
Actor Visual



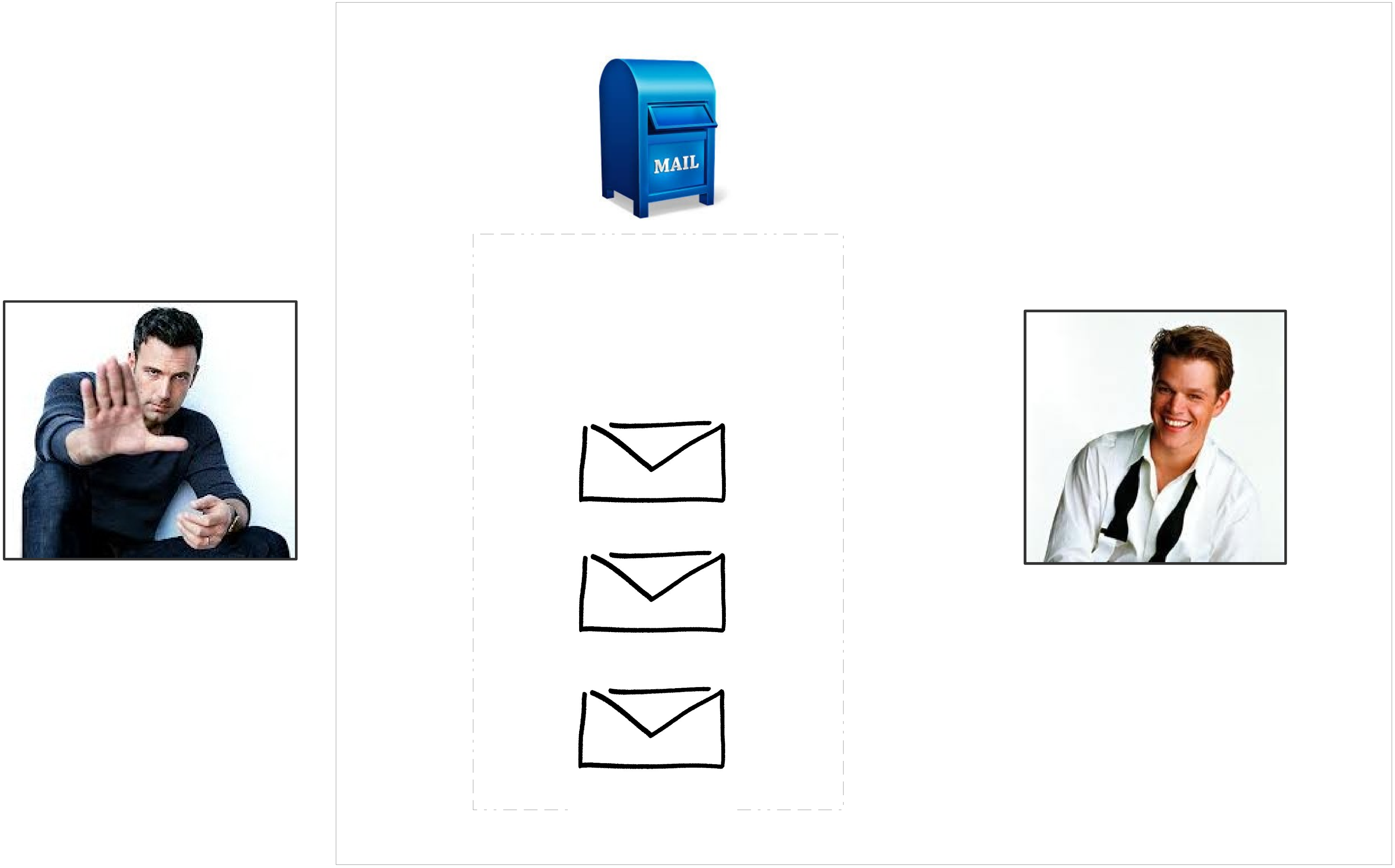
Actor Visual



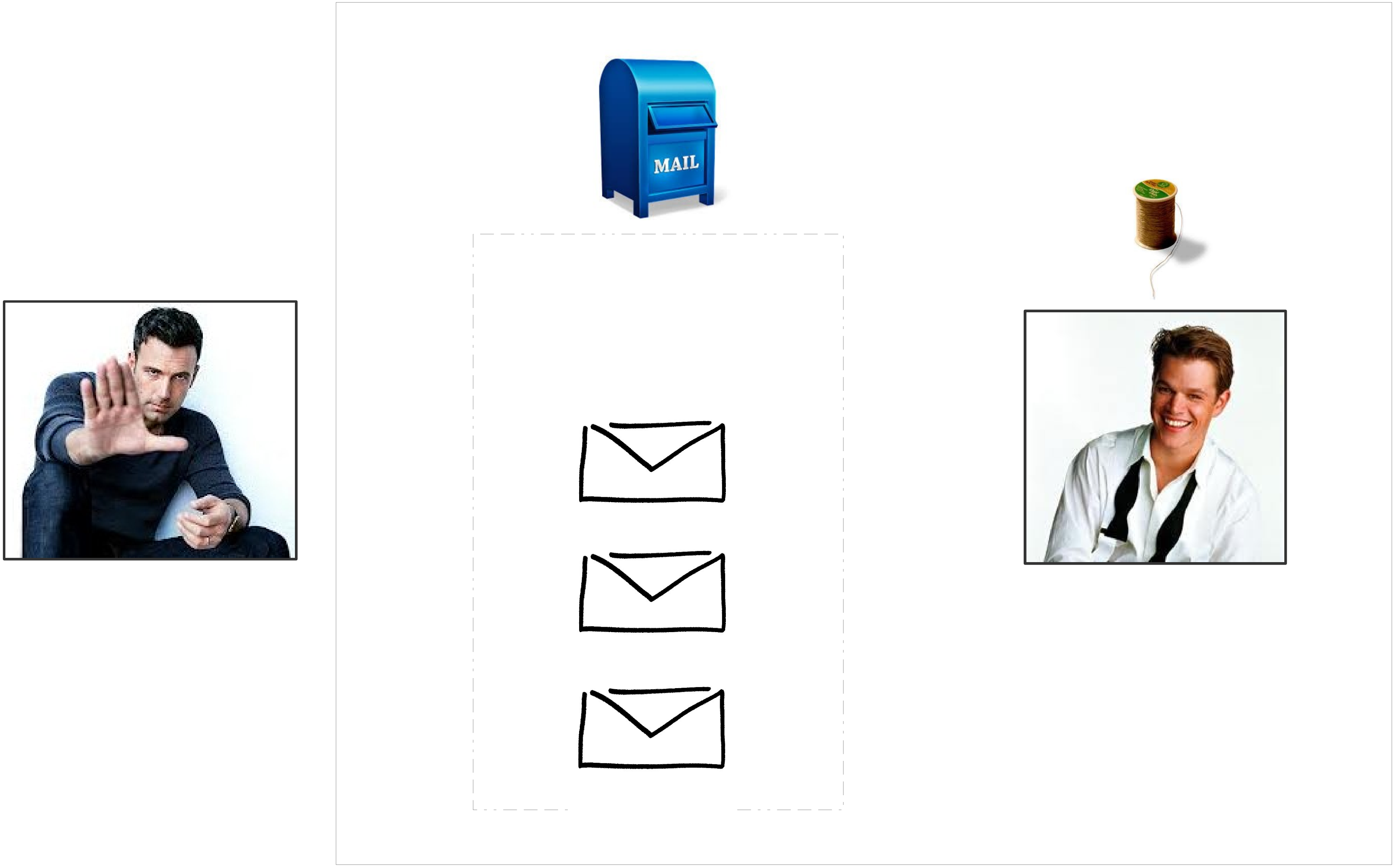
Actor Visual



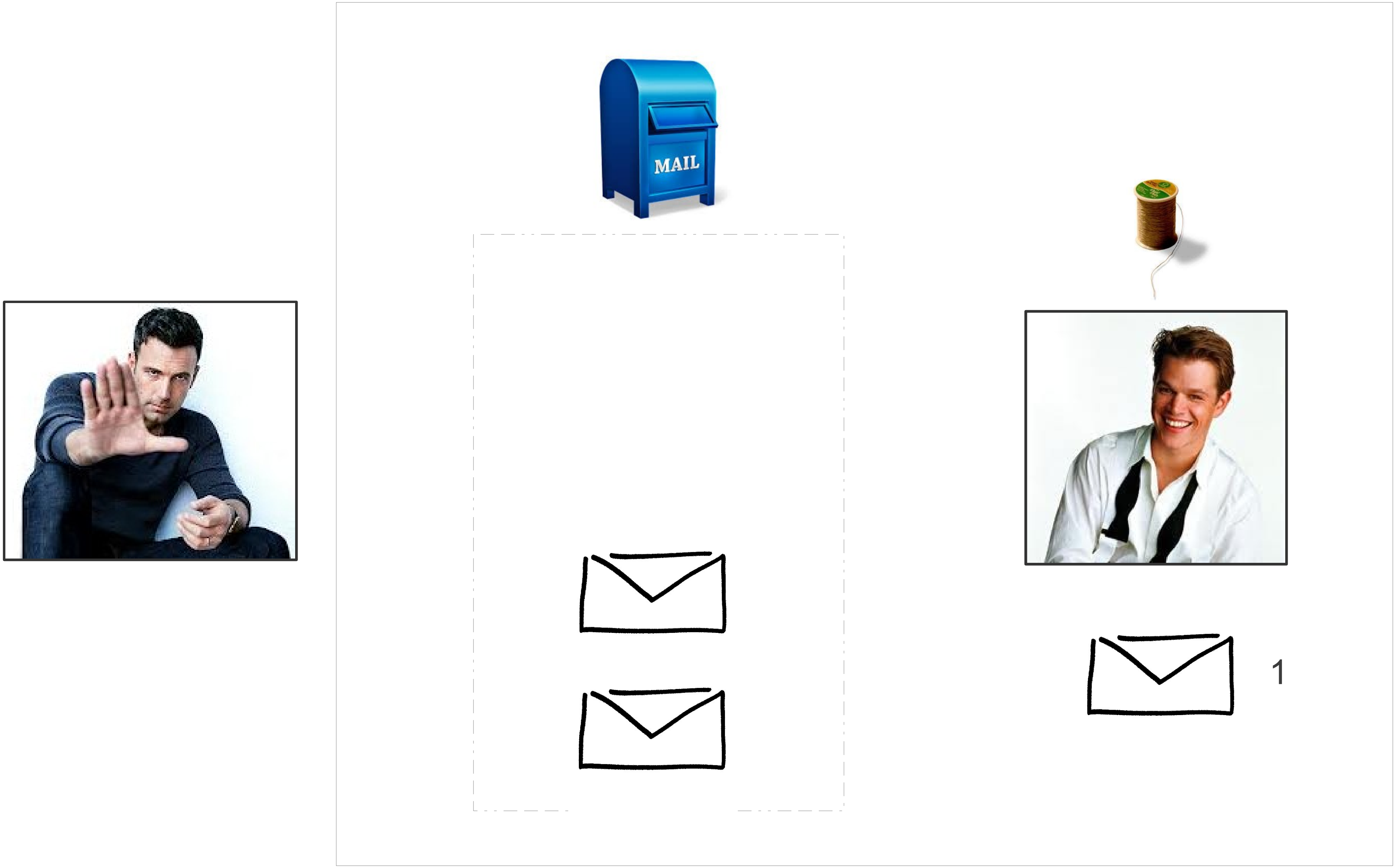
Actor Visual



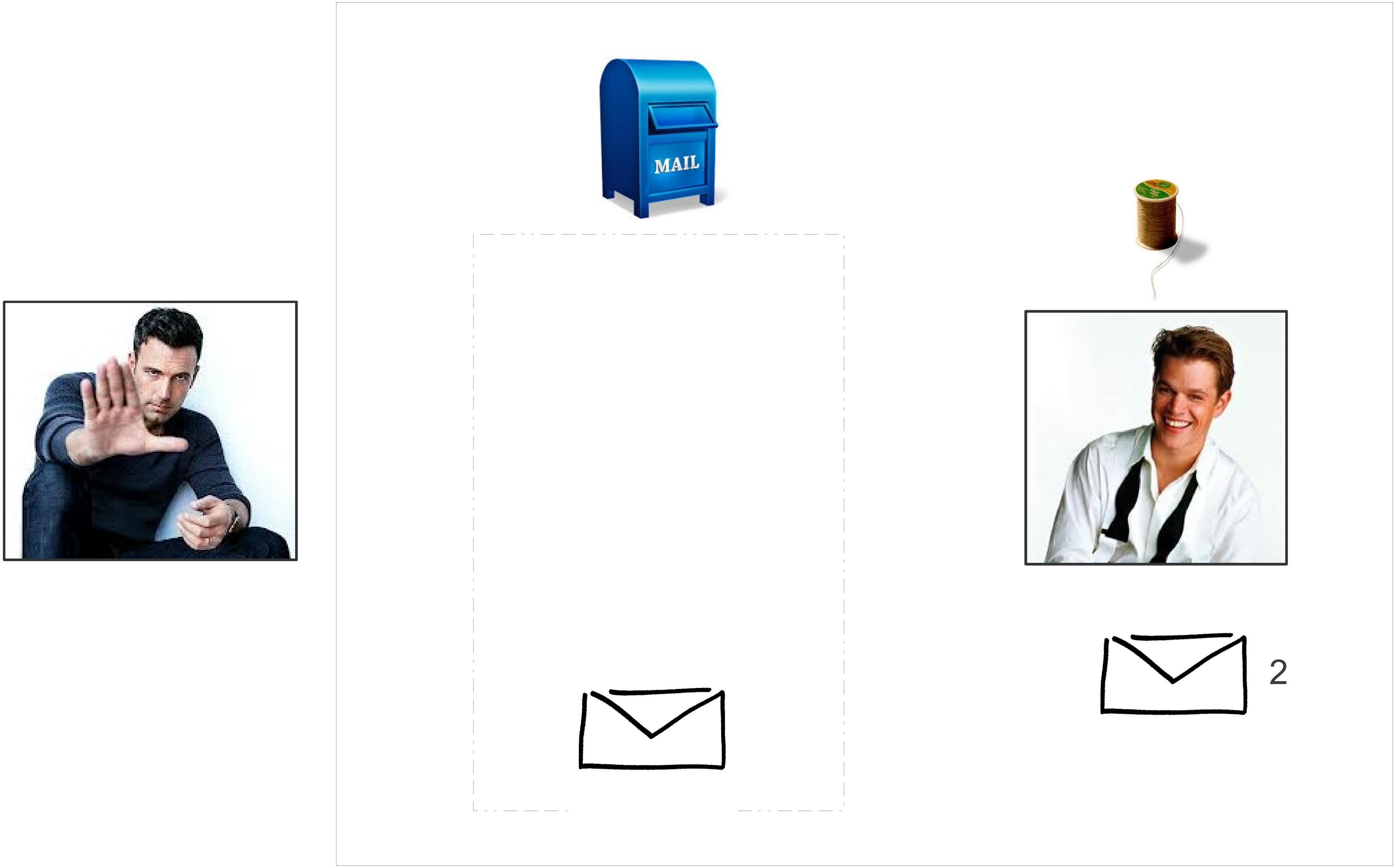
Actor Visual



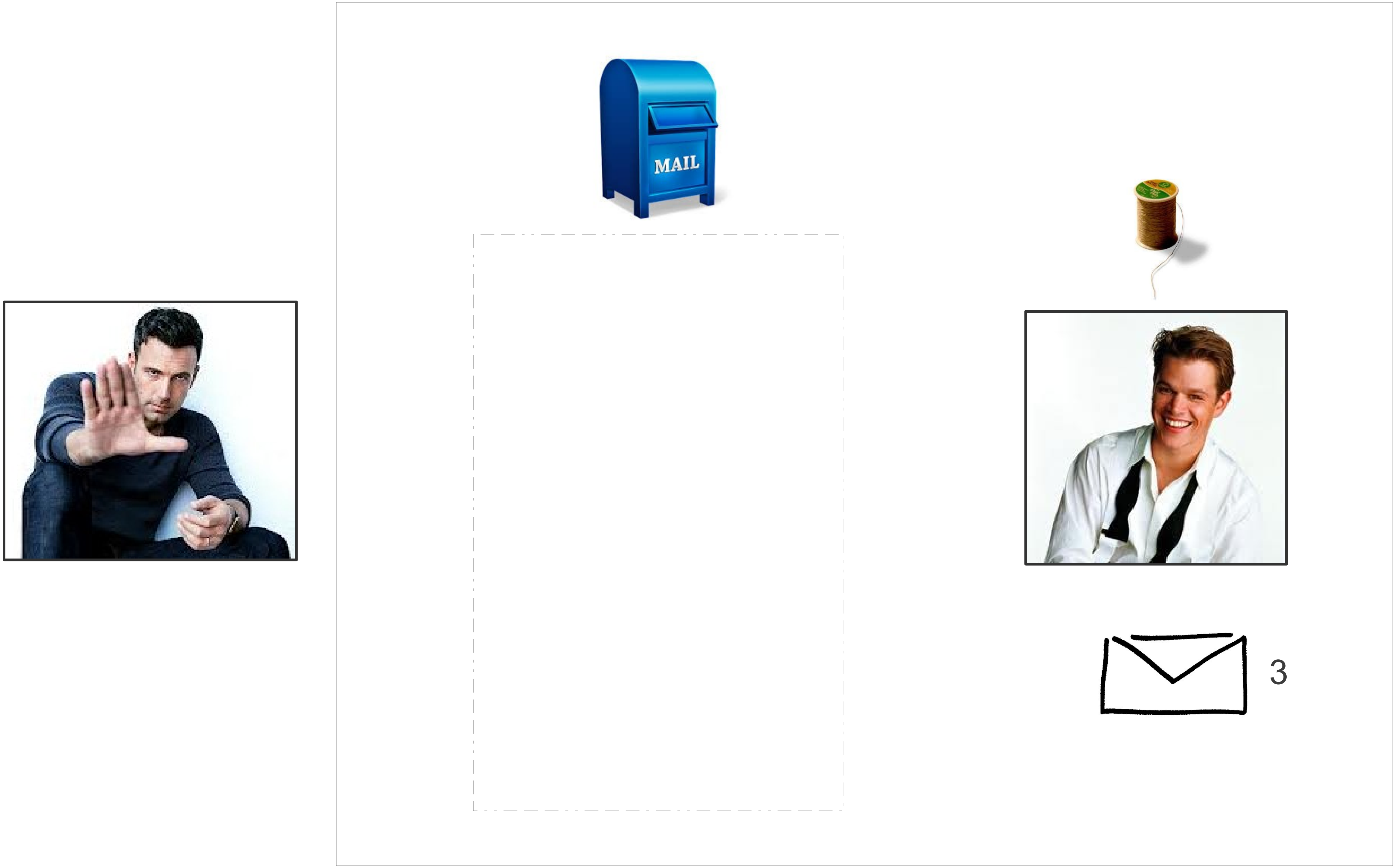
Actor Visual



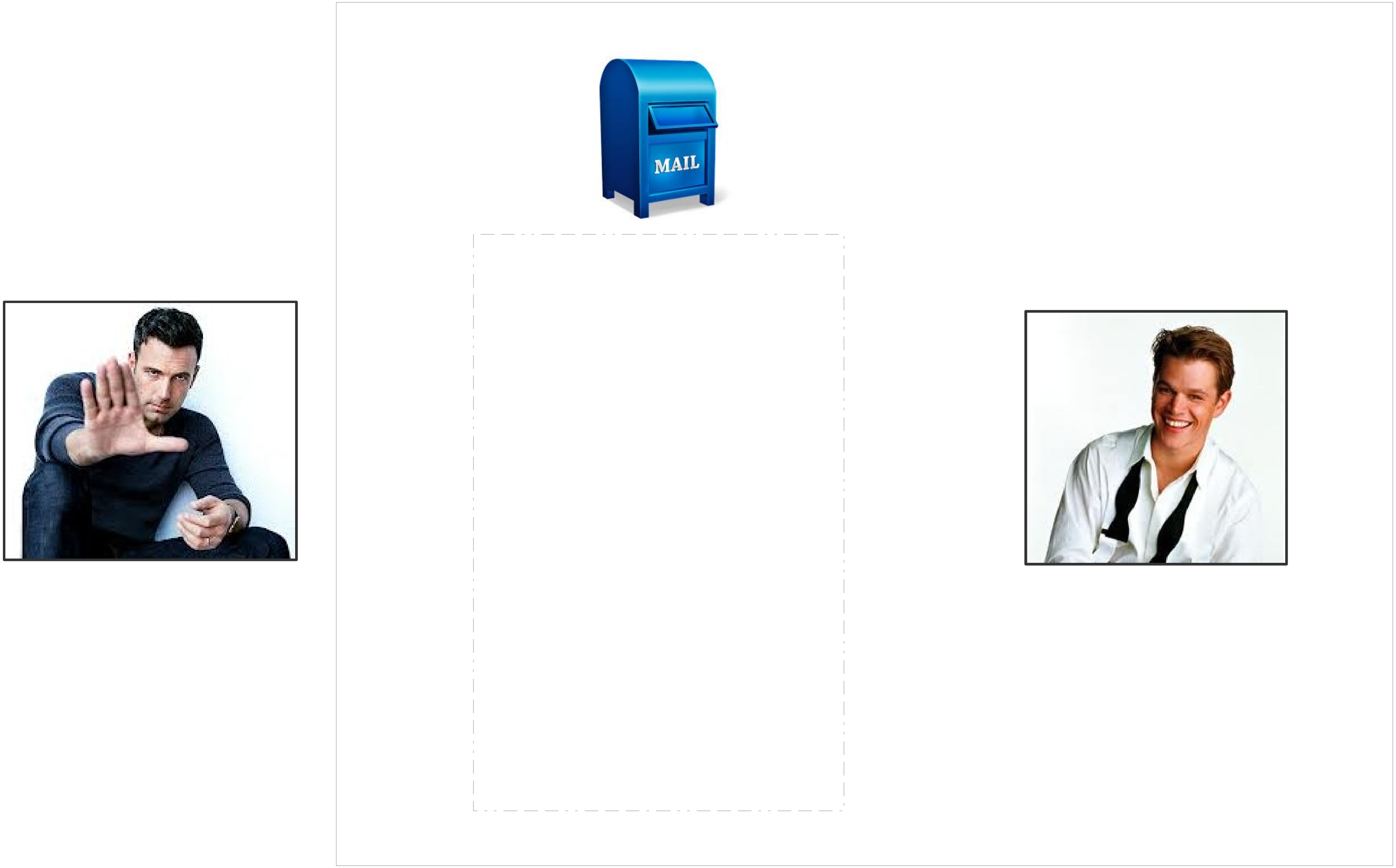
Actor Visual



Actor Visual



Actor Visual



Location Transparency

- Actors can be local and remote
- Vertical and Horizontal Growth
- Horizontal Growth driven by remoting systems
- Vertical Growth driven by routers
- All configuration based

Parralelism vs. Concurrency

- Parallelism: A condition that arises when at least two threads are executing simultaneously.
- Concurrency: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.
- Akka does whatever you tell it to do.

Untyped Actor in Java

```java
public class SimpleActorJava extends UntypedActor {
    LoggingAdapter log = Logging.getLogger
                         (getContext().system(), this);

    public void onReceive(Object message) {
        if (message instanceof String)
            log.info(
                "Received String message in SimpleActorJava {}",
                 message);
        else
            unhandled(message);
    }
}
```

Actor in Scala

```
class SimpleActorScala extends Actor {
   val log = Logging(context.system, this)

   def receive = {
     case "test" ⇒ log.info
                   ("received message test
                     in Simple Actor Scala")
     case _ ⇒ log.info("received unknown message
                        test in Simple Actor Scala")
   }
 }
```

Actor System

- Houses Untyped and Typed Actors
- Tasks are split up and delegated until they become small enough to be handled in one piece.
- The root of all Actors.

Actor System in Java

```
ActorSystem system = ActorSystem.create("MySystem");
```

Actor System in Scala

```
val system = ActorSystem("MySystem")
```

actorOf()

- Creates an actor onto a system
- Creates an actor with a name
- You will receive an `ActorRef` in return
- Using the `ActorRef` a message can be sent to the Actor that the `ActorRef` represents

actorOf() in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala], name =
                             "simpleActorJava")
```

actorOf() in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(new Props(SimpleActorJava.class),
"simpleActorJava");
```

Sending a message in Scala

```
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[SimpleActorScala], name =
                             "simpleActorJava")
myActor ! "Simple Test"
myActor ! "test"
```

Sending a message in Java

```
ActorSystem system = ActorSystem.create("MySystem");
ActorRef myActor = system.actorOf(new
                     Props(SimpleActorJava.class),
                           "simpleActorJava");
myActor.tell("Bueno!"); //Depracated. But works.
```

Demo of Running an Actor

Mad Props

- Specifies the creation of Actors

Mad Props in Java

```
ActorRef myActor = system.actorOf(new
                     Props(SimpleActorJava.class),
                           "simpleActorJava");


ActorRef myActor = system.actorOf(new
                     Props(
    new UntypedActorFactory() {
        public UntypedActor create() {
          return new SimpleActorJava();
    }}), "simpleActorJava");
```

Mad Props in Java

```
Props basic = new Props();

Props withActors = basic.withCreator(
      new UntypedActorFactory() {
          public UntypedActor create() {
             return new MyUntypedActor();
      }
});
```

Mad Props in Scala

```
val props1 = Props.empty
val props2 = Props[SimpleActorScala]
val props3 = Props(new SimpleActorScala)
val props4 = Props(
  creator = { () ⇒ new SimpleActorScala})
val props5 = props1.withCreator(new MyActor)
```

Actor Refs

- A subtype of `ActorRef`
- Intent is to send messages to `Actor` that it represents, proxy.
- `self()` refers to it's own referenc
- Each actor has reference to the `sender()` that sent the message.
- Any reference can be sent to another actor so that actor can send messages to it.

Various Other Types Of References

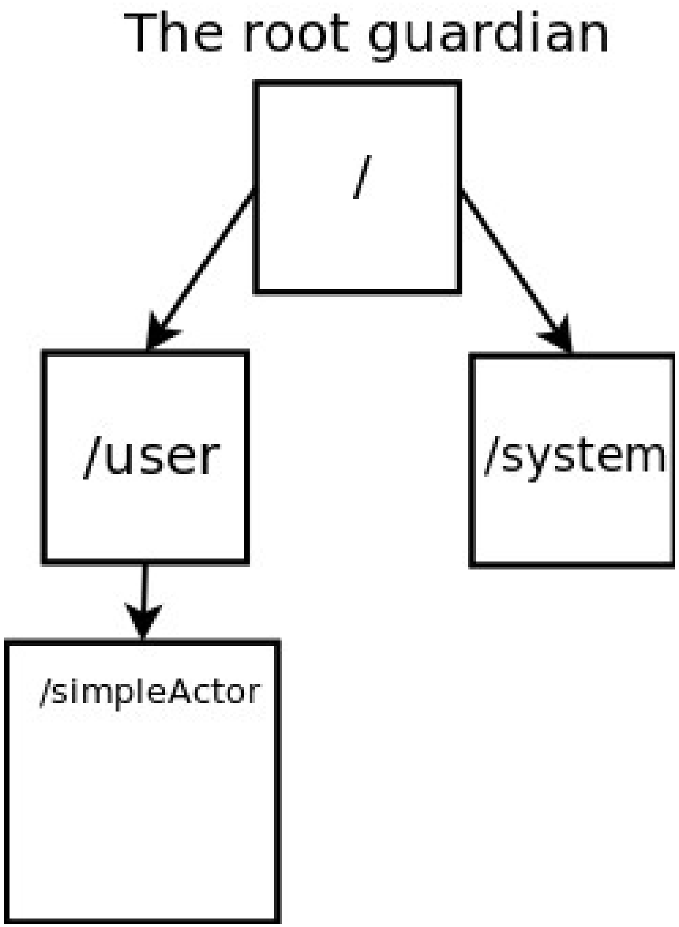
- Pure Local Actor References
- Local Actor References
- Local Actor References with Routing
- Remote Actor References
- Promise Actor References
- Dead Letter Actor References
- Cluster Actor References

Paths

- Can be local or remote
- Can be absolute or relative

```
"akka://my-sys/user/service-a/worker1" // purely local

"akka://my-sys@host.example.com:5678/user/service-b" // local or remote

"cluster://my-cluster/service-c" // clustered (Future Extension)
```

Anatomy of the Actor System



Logical/Physical Paths

- A logical path is seen as if one element is a without recognized if one actor is remote
- A physical path is the direct path to the server and the direct location.

actorFor()

- Actor references can be looked up using `ActorSystem.actorFor` method.
- `ActorSystem.actorFor` can return a local or a remote reference.
- Reference can be used as long as the actor is alive.
- Only ever looks up an existing actor, i.e. does not create one.
- For Local Actor References
  - The actor must exist
  - If not, you will receive an `EmptyLocalActorRef`
- For Remote Actor References
  - A search by path on the remote system will occur.

actorFor()

```
val system = ActorSystem("MySystem")

system.actorOf(Props[SimpleActorScala],
name = "simpleActorJava")

val myActor =
system.actorFor("akka://MySystem/user/simp
leActorJava")

myActor ! "Simple Test"
```

Typed Actor

- POJO and Actor Hybrids
- Consists of Two Parts, a public interface, and an implementation
- Cannot use `become`/`unbecome` to the type.
- Proxied
- Use Sparingly

Typed Actor in Java

```
public class Person { private String firstName; private String
lastName; //bunch of junk}


public interface RegistrationActor {
    public void registerPerson(Person person);

    public int getCount();
}
```

Typed Actor in Java

```
public class RegistrationActorImpl
      implements RegistrationActor, TypedActor.PreStart {

      private List<Person> people;

      @Override
      public void preStart() {
          this.people = new ArrayList<>();
      }

      @Override
      public void registerPerson(Person person) {
          this.people.add(person);
      }

      @Override
      public int getCount() {
          return people.size();
      }
  }
```

Running the Typed Actor in Java

```
ActorSystem system = ActorSystem.create("MySystem");
RegistrationActor registrationActor =
      TypedActor.get(system).typedActorOf(
            new
      TypedProps<RegistrationActorImpl>(RegistrationActor.class,
RegistrationActorImpl.class),"registrationActor");
      registrationActor.registerPerson(new Person("Abraham",
"Lincoln"));
```

Typed Actor in Scala

```
case class Person(firstName: String, lastName: String)

trait RegistrationActor {
  def registerPerson(person: Person)

  def getCount: Int
}

class RegistrationActorImpl extends RegistrationActor {
  var list = List[Person]()

  def registerPerson(person: Person) {
    list = list :+ person
  }

  def getCount: Int = list.size
}
```

Running the Typed Actor in Scala

```
val system = ActorSystem("MySystem")
val registrationActor =
TypedActor(system).typedActorOf(TypedProps[RegistrationActorImpl
], name = "registrationActor")
registrationActor.registerPerson(Person("Cesar", "Chavez"))
```

Demo of Typed Actors!

self() , sender(), context

Dead Letters Mailbox

- `/dev/null` for Akka!
- Where bad letters go to die.
- If a message cannot be delivered due to non existing actor, actor death, or other reasons, the message will be placed in the dead letters mailbox.
- To retrieve all dead letters pluck it from the event stream

EventStream

- Pub/Sub Stream of Events
- Both System and User Generated
- Subscribers are `ActorRefs`
- Channels are Classes and Events
- `EventStreams` employ SubchannelClassification (you will receive message of that type or subtype)
- The event stream is the main event bus of each actor system

Demo: What is going into the Dead Letters?

Futures and Promises

- Data structure used to retrieve the result of some concurrent operation.
- It can be retrieved “synchronously” (blocked) or “asynchronously” (unblocked)
- Futures need something called an `ExecutionContext` in scope in order to work

Futures by themselves, blocking

```
implicit val executionContext =
ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(1
2))

val future = Future {
   "Hello" + " " + "World"
}

implicit val timeout = Timeout(5 seconds)

println("Step 1")
val result = Await.result(future, timeout.duration) //blocking
println("Step 2: " + result)
```

Futures by themselves, nonblocking

```
implicit val executionContext =
ExecutionContext.fromExecutorService(Executors.newFixedThreadPool(1
2))

val future = Future {
  "Hello" + " " + "World"
}

future foreach (x => println("****" + x)) //asynchonous
println("running1")
println("running2")
println("running3")
```

Demo: Futures & Asking in Scala

Configuration

HOCON ANGRY!

HOCON

"Human-Optimized Config Object Notation"

HOCON

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

HOCON

```
db {
  default.driver=org.h2.Driver
  default.url="jdbc:h2:mem:play"
  default.user=sa
  default.password=""
}
```

HOCON

```
db {
  default{
    driver=org.h2.Driver
    url="jdbc:h2:mem:play"
    user=sa
    password=""
  }
}
```

application.conf

- Contains settings.
- Based on HOCON
- Can be set up in parts

Demo: Remote Actors

Demo: Routers

Demo: Becoming & Unbecoming

Thank You!