**Creating Actors**

**Note**

Since Akka enforces parental supervision every actor is supervised and (potentially) the supervisor of its children, it is advisable that you familiarize yourself with [*Actor Systems*](http://doc.akka.io/docs/akka/2.1.4/general/actor-systems.html#actor-systems) and [*Supervision and Monitoring*](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#supervision)and it may also help to read [*Summary: actorOf vs. actorFor*](http://doc.akka.io/docs/akka/2.1.4/general/addressing.html#actorof-vs-actorfor) (the whole of [*Actor References, Paths and Addresses*](http://doc.akka.io/docs/akka/2.1.4/general/addressing.html#addressing) is recommended reading in any case).

**Defining an Actor class**

Actor in Java are implemented by extending the UntypedActor class and implementing the onReceivemethod. This method takes the message as a parameter.

Here is an example:

1. import akka.actor.UntypedActor;
2. import akka.event.Logging;
3. import akka.event.LoggingAdapter;
5. public class MyUntypedActor extends UntypedActor {
6. LoggingAdapter log = Logging.getLogger(getContext().system(), this);
8. public void onReceive(Object message) throws Exception {
9. if (message instanceof String)
10. log.info("Received String message: {}", message);
11. else
12. unhandled(message);
13. }
14. }

**Props**

Props is a configuration class to specify options for the creation of actors. Here are some examples on how to create a Props instance.

1. Props props1 = new Props();
2. Props props2 = new Props(MyUntypedActor.class);
3. Props props3 = new Props(new UntypedActorFactory() {
4. public UntypedActor create() {
5. return new MyUntypedActor();
6. }
7. });
8. Props props4 = props1.withCreator(new UntypedActorFactory() {
9. public UntypedActor create() {
10. return new MyUntypedActor();
11. }
12. });

**Creating Actors with Props**

Actors are created by passing in a Props instance into the actorOf factory method.

1. ActorRef myActor = system.actorOf(
2. new Props(MyUntypedActor.class).withDispatcher("my-dispatcher"), "myactor");

**Creating Actors with default constructor**

1. import akka.actor.ActorRef;
2. import akka.actor.ActorSystem;
3. import akka.actor.Props;
4. ActorSystem system = ActorSystem.create("MySystem");
5. ActorRef myActor = system.actorOf(new Props(MyUntypedActor.class), "myactor");

The call to actorOf returns an instance of ActorRef. This is a handle to the UntypedActor instance which you can use to interact with the UntypedActor. The ActorRef is immutable and has a one to one relationship with the Actor it represents. The ActorRef is also serializable and network-aware. This means that you can serialize it, send it over the wire and use it on a remote host and it will still be representing the same Actor on the original node, across the network.

In the above example the actor was created from the system. It is also possible to create actors from other actors with the actor context. The difference is how the supervisor hierarchy is arranged. When using the context the current actor will be supervisor of the created child actor. When using the system it will be a top level actor, that is supervised by the system (internal guardian actor).

1. public class FirstUntypedActor extends UntypedActor {
2. ActorRef myActor = getContext().actorOf(new Props(MyActor.class), "myactor");

The name parameter is optional, but you should preferably name your actors, since that is used in log messages and for identifying actors. The name must not be empty or start with $, but it may contain URL encoded characters (eg. %20 for a blank space). If the given name is already in use by another child to the same parent actor an InvalidActorNameException is thrown.

Actors are automatically started asynchronously when created. When you create the UntypedActor then it will automatically call the preStart callback method on the UntypedActor class. This is an excellent place to add initialization code for the actor.

1. @Override
2. public void preStart() {
3. ... // initialization code
4. }

**Creating Actors with non-default constructor**

If your UntypedActor has a constructor that takes parameters then you can't create it using 'actorOf(new Props(clazz))'. Then you can instead pass in 'new Props(new UntypedActorFactory() {..})' in which you can create the Actor in any way you like.

Here is an example:

1. // allows passing in arguments to the MyActor constructor
2. ActorRef myActor = system.actorOf(new Props(new UntypedActorFactory() {
3. public UntypedActor create() {
4. return new MyActor("...");
5. }
6. }), "myactor");

This way of creating the Actor is also great for integrating with Dependency Injection (DI) frameworks like Guice or Spring.

**Warning**

You might be tempted at times to offer an UntypedActor factory which always returns the same instance, e.g. by using a static field. This is not supported, as it goes against the meaning of an actor restart, which is described here: [*What Restarting Means*](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#supervision-restart).

**UntypedActor API**

The UntypedActor class defines only one abstract method, the above mentioned onReceive(Objectmessage), which implements the behavior of the actor.

If the current actor behavior does not match a received message, it's recommended that you call theunhandled method, which by default publishes a new akka.actor.UnhandledMessage(message, sender,recipient) on the actor system’s event stream (set configuration item akka.actor.debug.unhandled to on to have them converted into actual Debug messages).

In addition, it offers:

* getSelf() reference to the ActorRef of the actor
* getSender() reference sender Actor of the last received message, typically used as described in [*Reply to messages*](http://doc.akka.io/docs/akka/2.1.4/java/untyped-actors.html#untypedactor-reply)
* supervisorStrategy() user overridable definition the strategy to use for supervising child actors

This strategy is typically declared inside the actor in order to have access to the actor’s internal state within the decider function: since failure is communicated as a message sent to the supervisor and processed like other messages (albeit outside of the normal behavior), all values and variables within the actor are available, as is the getSender() reference (which will be the immediate child reporting the failure; if the original failure occurred within a distant descendant it is still reported one level up at a time).

* getContext() exposes contextual information for the actor and the current message, such as:
  + factory methods to create child actors (actorOf)
  + system that the actor belongs to
  + parent supervisor
  + supervised children
  + lifecycle monitoring
  + hotswap behavior stack as described in *[HotSwap](http://doc.akka.io/docs/akka/2.1.4/java/untyped-actors.html" \l "untypedactor-hotswap)*

The remaining visible methods are user-overridable life-cycle hooks which are described in the following:

1. public void preStart() {
2. }
4. public void preRestart(Throwable reason, Option<Object> message) {
5. for (ActorRef each : getContext().getChildren())
6. getContext().stop(each);
7. postStop();
8. }
10. public void postRestart(Throwable reason) {
11. preStart();
12. }
14. public void postStop() {
15. }

The implementations shown above are the defaults provided by the UntypedActor class.

**Lifecycle Monitoring aka DeathWatch**

In order to be notified when another actor terminates (i.e. stops permanently, not temporary failure and restart), an actor may register itself for reception of the Terminated message dispatched by the other actor upon termination (see [Stopping Actors](http://doc.akka.io/docs/akka/2.1.4/java/untyped-actors.html#stopping-actors)). This service is provided by the DeathWatch component of the actor system.

Registering a monitor is easy (see fourth line, the rest is for demonstrating the whole functionality):

1. public class WatchActor extends UntypedActor {
2. final ActorRef child = this.getContext().actorOf(Props.empty(), "child");
3. {
4. this.getContext().watch(child); // <-- the only call needed for registration
5. }
6. ActorRef lastSender = getContext().system().deadLetters();
8. @Override
9. public void onReceive(Object message) {
10. if (message.equals("kill")) {
11. getContext().stop(child);
12. lastSender = getSender();
13. } else if (message instanceof Terminated) {
14. final Terminated t = (Terminated) message;
15. if (t.getActor() == child) {
16. lastSender.tell("finished", getSelf());
17. }
18. } else {
19. unhandled(message);
20. }
21. }
22. }

It should be noted that the Terminated message is generated independent of the order in which registration and termination occur. Registering multiple times does not necessarily lead to multiple messages being generated, but there is no guarantee that only exactly one such message is received: if termination of the watched actor has generated and queued the message, and another registration is done before this message has been processed, then a second message will be queued, because registering for monitoring of an already terminated actor leads to the immediate generation of the Terminated message.

It is also possible to deregister from watching another actor’s liveliness using context.unwatch(target), but obviously this cannot guarantee non-reception of the Terminated message because that may already have been queued.

**Start Hook**

Right after starting the actor, its preStart method is invoked.

1. @Override
2. public void preStart() {
3. // registering with other actors
4. someService.tell(Register(getSelf());
5. }

**Restart Hooks**

All actors are supervised, i.e. linked to another actor with a fault handling strategy. Actors may be restarted in case an exception is thrown while processing a message (see [*Supervision and Monitoring*](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#supervision)). This restart involves the hooks mentioned above:

1. The old actor is informed by calling preRestart with the exception which caused the restart and the message which triggered that exception; the latter may be None if the restart was not caused by processing a message, e.g. when a supervisor does not trap the exception and is restarted in turn by its supervisor, or if an actor is restarted due to a sibling’s failure. If the message is available, then that message’s sender is also accessible in the usual way (i.e. by calling getSender()).

This method is the best place for cleaning up, preparing hand-over to the fresh actor instance, etc. By default it stops all children and calls postStop.

1. The initial factory from the actorOf call is used to produce the fresh instance.
2. The new actor’s postRestart method is invoked with the exception which caused the restart. By default the preStart is called, just as in the normal start-up case.

An actor restart replaces only the actual actor object; the contents of the mailbox is unaffected by the restart, so processing of messages will resume after the postRestart hook returns. The message that triggered the exception will not be received again. Any message sent to an actor while it is being restarted will be queued to its mailbox as usual.

**Stop Hook**

After stopping an actor, its postStop hook is called, which may be used e.g. for deregistering this actor from other services. This hook is guaranteed to run after message queuing has been disabled for this actor, i.e. messages sent to a stopped actor will be redirected to the deadLetters of the ActorSystem.

**Identifying Actors**

As described in [*Actor References, Paths and Addresses*](http://doc.akka.io/docs/akka/2.1.4/general/addressing.html#addressing), each actor has a unique logical path, which is obtained by following the chain of actors from child to parent until reaching the root of the actor system, and it has a physical path, which may differ if the supervision chain includes any remote supervisors. These paths are used by the system to look up actors, e.g. when a remote message is received and the recipient is searched, but they are also useful more directly: actors may look up other actors by specifying absolute or relative paths—logical or physical—and receive back an ActorRef with the result:

1. getContext().actorFor("/user/serviceA/actor") // will look up this absolute path
2. getContext().actorFor("../joe") // will look up sibling beneath same supervisor

The supplied path is parsed as a java.net.URI, which basically means that it is split on / into path elements. If the path starts with /, it is absolute and the look-up starts at the root guardian (which is the parent of"/user"); otherwise it starts at the current actor. If a path element equals .., the look-up will take a step “up” towards the supervisor of the currently traversed actor, otherwise it will step “down” to the named child. It should be noted that the .. in actor paths here always means the logical structure, i.e. the supervisor.

If the path being looked up does not exist, a special actor reference is returned which behaves like the actor system’s dead letter queue but retains its identity (i.e. the path which was looked up).

Remote actor addresses may also be looked up, if remoting is enabled:

1. getContext().actorFor("akka://app@otherhost:1234/user/serviceB")

These look-ups return a (possibly remote) actor reference immediately, so you will have to send to it and await a reply in order to verify that serviceB is actually reachable and running. An example demonstrating actor look-up is given in [*Remote Lookup*](http://doc.akka.io/docs/akka/2.1.4/java/remoting.html#remote-lookup-sample-java).

**Messages and immutability**

**IMPORTANT**: Messages can be any kind of object but have to be immutable. Akka can’t enforce immutability (yet) so this has to be by convention.

Here is an example of an immutable message:

1. public class ImmutableMessage {
2. private final int sequenceNumber;
3. private final List<String> values;
5. public ImmutableMessage(int sequenceNumber, List<String> values) {
6. this.sequenceNumber = sequenceNumber;
7. this.values = Collections.unmodifiableList(new ArrayList<String>(values));
8. }
10. public int getSequenceNumber() {
11. return sequenceNumber;
12. }
14. public List<String> getValues() {
15. return values;
16. }
17. }

**Send messages**

Messages are sent to an Actor through one of the following methods.

* tell means “fire-and-forget”, e.g. send a message asynchronously and return immediately.
* ask sends a message asynchronously and returns a Future representing a possible reply.

Message ordering is guaranteed on a per-sender basis.

**Note**

There are performance implications of using ask since something needs to keep track of when it times out, there needs to be something that bridges a Promise into an ActorRef and it also needs to be reachable through remoting. So always prefer tell for performance, and only ask if you must.

In all these methods you have the option of passing along your own ActorRef. Make it a practice of doing so because it will allow the receiver actors to be able to respond to your message, since the sender reference is sent along with the message.

**Tell: Fire-forget**

This is the preferred way of sending messages. No blocking waiting for a message. This gives the best concurrency and scalability characteristics.

1. actor.tell("Hello");

Or with the sender reference passed along with the message and available to the receiving Actor in itsgetSender: ActorRef member field. The target actor can use this to reply to the original sender, by usinggetSender().tell(replyMsg).

1. actor.tell("Hello", getSelf());

If invoked without the sender parameter the sender will be deadLetters actor reference in the target actor.

**Ask: Send-And-Receive-Future**

The ask pattern involves actors as well as futures, hence it is offered as a use pattern rather than a method on ActorRef:

1. import static akka.pattern.Patterns.ask;
2. import static akka.pattern.Patterns.pipe;
3. import scala.concurrent.Future;
4. import akka.dispatch.Futures;
5. import scala.concurrent.duration.Duration;
6. import akka.util.Timeout;
7. import java.util.concurrent.TimeUnit;
8. import java.util.ArrayList;
9. final Timeout t = new Timeout(Duration.create(5, TimeUnit.SECONDS));
11. final ArrayList<Future<Object>> futures = new ArrayList<Future<Object>>();
12. futures.add(ask(actorA, "request", 1000)); // using 1000ms timeout
13. futures.add(ask(actorB, "another request", t)); // using timeout from above
15. final Future<Iterable<Object>> aggregate =
16. Futures.sequence(futures, system.dispatcher());
18. final Future<Result> transformed = aggregate.map(
19. new Mapper<Iterable<Object>, Result>() {
20. public Result apply(Iterable<Object> coll) {
21. final Iterator<Object> it = coll.iterator();
22. final String s = (String) it.next();
23. final int x = (Integer) it.next();
24. return new Result(x, s);
25. }
26. }, system.dispatcher());
28. pipe(transformed, system.dispatcher()).to(actorC);

This example demonstrates ask together with the pipe pattern on futures, because this is likely to be a common combination. Please note that all of the above is completely non-blocking and asynchronous: askproduces a Future, two of which are composed into a new future using the Futures.sequence and mapmethods and then pipe installs an onComplete-handler on the future to effect the submission of the aggregated Result to another actor.

Using ask will send a message to the receiving Actor as with tell, and the receiving actor must reply withgetSender().tell(reply) in order to complete the returned Future with a value. The ask operation involves creating an internal actor for handling this reply, which needs to have a timeout after which it is destroyed in order not to leak resources; see more below.

**Warning**

To complete the future with an exception you need send a Failure message to the sender. This is *not done automatically* when an actor throws an exception while processing a message.

1. try {
2. String result = operation();
3. getSender().tell(result, getSelf());
4. } catch (Exception e) {
5. getSender().tell(new akka.actor.Status.Failure(e), getSelf());
6. throw e;
7. }

If the actor does not complete the future, it will expire after the timeout period, specified as parameter to theask method; this will complete the Future with an AskTimeoutException.

See [*Futures (Java)*](http://doc.akka.io/docs/akka/2.1.4/java/futures.html#futures-java) for more information on how to await or query a future.

The onComplete, onSuccess, or onFailure methods of the Future can be used to register a callback to get a notification when the Future completes. Gives you a way to avoid blocking.

**Warning**

When using future callbacks, inside actors you need to carefully avoid closing over the containing actor’s reference, i.e. do not call methods or access mutable state on the enclosing actor from within the callback. This would break the actor encapsulation and may introduce synchronization bugs and race conditions because the callback will be scheduled concurrently to the enclosing actor. Unfortunately there is not yet a way to detect these illegal accesses at compile time. See also: [*Actors and shared mutable state*](http://doc.akka.io/docs/akka/2.1.4/general/jmm.html#jmm-shared-state)

**Forward message**

You can forward a message from one actor to another. This means that the original sender address/reference is maintained even though the message is going through a 'mediator'. This can be useful when writing actors that work as routers, load-balancers, replicators etc. You need to pass along your context variable as well.

1. myActor.forward(message, getContext());

**Receive messages**

When an actor receives a message it is passed into the onReceive method, this is an abstract method on theUntypedActor base class that needs to be defined.

Here is an example:

1. import akka.actor.UntypedActor;
2. import akka.event.Logging;
3. import akka.event.LoggingAdapter;
5. public class MyUntypedActor extends UntypedActor {
6. LoggingAdapter log = Logging.getLogger(getContext().system(), this);
8. public void onReceive(Object message) throws Exception {
9. if (message instanceof String)
10. log.info("Received String message: {}", message);
11. else
12. unhandled(message);
13. }
14. }

An alternative to using if-instanceof checks is to use [Apache Commons MethodUtils](http://commons.apache.org/beanutils/api/org/apache/commons/beanutils/MethodUtils.html#invokeMethod(java.lang.Object,%20java.lang.String,%20java.lang.Object)) to invoke a named method whose parameter type matches the message type.

**Reply to messages**

If you want to have a handle for replying to a message, you can use getSender(), which gives you an ActorRef. You can reply by sending to that ActorRef with getSender().tell(replyMsg). You can also store the ActorRef for replying later, or passing on to other actors. If there is no sender (a message was sent without an actor or future context) then the sender defaults to a 'dead-letter' actor ref.

1. public void onReceive(Object request) {
2. String result = process(request);
3. getSender().tell(result); // will have dead-letter actor as default
4. }

**Receive timeout**

The UntypedActorContext setReceiveTimeout defines the inactivity timeout after which the sending of aReceiveTimeout message is triggered. When specified, the receive function should be able to handle anakka.actor.ReceiveTimeout message. 1 millisecond is the minimum supported timeout.

Please note that the receive timeout might fire and enqueue the ReceiveTimeout message right after another message was enqueued; hence it is **not guaranteed** that upon reception of the receive timeout there must have been an idle period beforehand as configured via this method.

Once set, the receive timeout stays in effect (i.e. continues firing repeatedly after inactivity periods). Pass inDuration.Undefined to switch off this feature.

1. import akka.actor.ReceiveTimeout;
2. import akka.actor.UntypedActor;
3. import scala.concurrent.duration.Duration;
5. public class MyReceivedTimeoutUntypedActor extends UntypedActor {
7. public MyReceivedTimeoutUntypedActor() {
8. // To set an initial delay
9. getContext().setReceiveTimeout(Duration.create("30 seconds"));
10. }
12. public void onReceive(Object message) {
13. if (message.equals("Hello")) {
14. // To set in a response to a message
15. getContext().setReceiveTimeout(Duration.create("10 seconds"));
16. getSender().tell("Hello world", getSelf());
17. } else if (message == ReceiveTimeout.getInstance()) {
18. // To turn it off
19. getContext().setReceiveTimeout(Duration.Undefined());
20. throw new RuntimeException("received timeout");
21. } else {
22. unhandled(message);
23. }
24. }
25. }

**Stopping actors**

Actors are stopped by invoking the stop method of a ActorRefFactory, i.e. ActorContext or ActorSystem. Typically the context is used for stopping child actors and the system for stopping top level actors. The actual termination of the actor is performed asynchronously, i.e. stop may return before the actor is stopped.

Processing of the current message, if any, will continue before the actor is stopped, but additional messages in the mailbox will not be processed. By default these messages are sent to the deadLetters of theActorSystem, but that depends on the mailbox implementation.

Termination of an actor proceeds in two steps: first the actor suspends its mailbox processing and sends a stop command to all its children, then it keeps processing the termination messages from its children until the last one is gone, finally terminating itself (invoking postStop, dumping mailbox, publishing Terminated on the[*DeathWatch*](http://doc.akka.io/docs/akka/2.1.4/java/untyped-actors.html#deathwatch-java), telling its supervisor). This procedure ensures that actor system sub-trees terminate in an orderly fashion, propagating the stop command to the leaves and collecting their confirmation back to the stopped supervisor. If one of the actors does not respond (i.e. processing a message for extended periods of time and therefore not receiving the stop command), this whole process will be stuck.

Upon ActorSystem.shutdown, the system guardian actors will be stopped, and the aforementioned process will ensure proper termination of the whole system.

The postStop hook is invoked after an actor is fully stopped. This enables cleaning up of resources:

1. @Override
2. public void postStop() {
3. // close some file or database connection
4. }

**Note**

Since stopping an actor is asynchronous, you cannot immediately reuse the name of the child you just stopped; this will result in an InvalidActorNameException. Instead, watch the terminating actor and create its replacement in response to the Terminated message which will eventually arrive.

**PoisonPill**

You can also send an actor the akka.actor.PoisonPill message, which will stop the actor when the message is processed. PoisonPill is enqueued as ordinary messages and will be handled after messages that were already queued in the mailbox.

Use it like this:

1. import akka.actor.PoisonPill;
2. import akka.actor.Kill;
3. myActor.tell(PoisonPill.getInstance(), null);

**Graceful Stop**

gracefulStop is useful if you need to wait for termination or compose ordered termination of several actors:

1. import static akka.pattern.Patterns.gracefulStop;
2. import scala.concurrent.Future;
3. import scala.concurrent.Await;
4. import scala.concurrent.duration.Duration;
5. import akka.pattern.AskTimeoutException;
6. try {
7. Future<Boolean> stopped =
8. gracefulStop(actorRef, Duration.create(5, TimeUnit.SECONDS), system);
9. Await.result(stopped, Duration.create(6, TimeUnit.SECONDS));
10. // the actor has been stopped
11. } catch (AskTimeoutException e) {
12. // the actor wasn't stopped within 5 seconds
13. }

When gracefulStop() returns successfully, the actor’s postStop() hook will have been executed: there exists a happens-before edge between the end of postStop() and the return of gracefulStop().

**Warning**

Keep in mind that an actor stopping and its name being deregistered are separate events which happen asynchronously from each other. Therefore it may be that you will find the name still in use aftergracefulStop() returned. In order to guarantee proper deregistration, only reuse names from within a supervisor you control and only in response to a Terminated message, i.e. not for top-level actors.

**HotSwap**

**Upgrade**

Akka supports hotswapping the Actor’s message loop (e.g. its implementation) at runtime. Use thegetContext().become method from within the Actor. The hotswapped code is kept in a Stack which can be pushed (replacing or adding at the top) and popped.

**Warning**

Please note that the actor will revert to its original behavior when restarted by its Supervisor.

To hotswap the Actor using getContext().become:

1. import akka.japi.Procedure;
2. public class HotSwapActor extends UntypedActor {
4. Procedure<Object> angry = new Procedure<Object>() {
5. @Override
6. public void apply(Object message) {
7. if (message.equals("bar")) {
8. getSender().tell("I am already angry?", getSelf());
9. } else if (message.equals("foo")) {
10. getContext().become(happy);
11. }
12. }
13. };
15. Procedure<Object> happy = new Procedure<Object>() {
16. @Override
17. public void apply(Object message) {
18. if (message.equals("bar")) {
19. getSender().tell("I am already happy :-)", getSelf());
20. } else if (message.equals("foo")) {
21. getContext().become(angry);
22. }
23. }
24. };
26. public void onReceive(Object message) {
27. if (message.equals("bar")) {
28. getContext().become(angry);
29. } else if (message.equals("foo")) {
30. getContext().become(happy);
31. } else {
32. unhandled(message);
33. }
34. }
35. }

This variant of the become method is useful for many different things, such as to implement a Finite State Machine (FSM). It will replace the current behavior (i.e. the top of the behavior stack), which means that you do not use unbecome, instead always the next behavior is explicitly installed.

The other way of using become does not replace but add to the top of the behavior stack. In this case care must be taken to ensure that the number of “pop” operations (i.e. unbecome) matches the number of “push” ones in the long run, otherwise this amounts to a memory leak (which is why this behavior is not the default).

1. public class UntypedActorSwapper {
3. public static class Swap {
4. public static Swap SWAP = new Swap();
6. private Swap() {
7. }
8. }
10. public static class Swapper extends UntypedActor {
11. LoggingAdapter log = Logging.getLogger(getContext().system(), this);
13. public void onReceive(Object message) {
14. if (message == SWAP) {
15. log.info("Hi");
16. getContext().become(new Procedure<Object>() {
17. @Override
18. public void apply(Object message) {
19. log.info("Ho");
20. getContext().unbecome(); // resets the latest 'become'
21. }
22. }, false); // this signals stacking of the new behavior
23. } else {
24. unhandled(message);
25. }
26. }
27. }
29. public static void main(String... args) {
30. ActorSystem system = ActorSystem.create("MySystem");
31. ActorRef swap = system.actorOf(new Props(Swapper.class));
32. swap.tell(SWAP, null); // logs Hi
33. swap.tell(SWAP, null); // logs Ho
34. swap.tell(SWAP, null); // logs Hi
35. swap.tell(SWAP, null); // logs Ho
36. swap.tell(SWAP, null); // logs Hi
37. swap.tell(SWAP, null); // logs Ho
38. }
40. }

**Stash**

The UntypedActorWithStash class enables an actor to temporarily stash away messages that can not or should not be handled using the actor's current behavior. Upon changing the actor's message handler, i.e., right before invoking getContext().become() or getContext().unbecome(), all stashed messages can be "unstashed", thereby prepending them to the actor's mailbox. This way, the stashed messages can be processed in the same order as they have been received originally.

**Warning**

Please note that the stash can only be used together with actors that have a deque-based mailbox. For this, configure the mailbox-type of the dispatcher to be a deque-based mailbox, such asakka.dispatch.UnboundedDequeBasedMailbox (see [*Dispatchers (Java)*](http://doc.akka.io/docs/akka/2.1.4/java/dispatchers.html#dispatchers-java)).

Here is an example of the UntypedActorWithStash class in action:

1. public class ActorWithProtocol extends UntypedActorWithStash {
2. public void onReceive(Object msg) {
3. if (msg.equals("open")) {
4. unstashAll();
5. getContext().become(new Procedure<Object>() {
6. public void apply(Object msg) throws Exception {
7. if (msg.equals("write")) {
8. // do writing...
9. } else if (msg.equals("close")) {
10. unstashAll();
11. getContext().unbecome();
12. } else {
13. stash();
14. }
15. }
16. }, false); // add behavior on top instead of replacing
17. } else {
18. stash();
19. }
20. }
21. }

Invoking stash() adds the current message (the message that the actor received last) to the actor's stash. It is typically invoked when handling the default case in the actor's message handler to stash messages that aren't handled by the other cases. It is illegal to stash the same message twice; to do so results in anIllegalStateException being thrown. The stash may also be bounded in which case invoking stash() may lead to a capacity violation, which results in a StashOverflowException. The capacity of the stash can be configured using the stash-capacity setting (an Int) of the dispatcher's configuration.

Invoking unstashAll() enqueues messages from the stash to the actor's mailbox until the capacity of the mailbox (if any) has been reached (note that messages from the stash are prepended to the mailbox). In case a bounded mailbox overflows, a MessageQueueAppendFailedException is thrown. The stash is guaranteed to be empty after calling unstashAll().

The stash is backed by a scala.collection.immutable.Vector. As a result, even a very large number of messages may be stashed without a major impact on performance.

Note that the stash is part of the ephemeral actor state, unlike the mailbox. Therefore, it should be managed like other parts of the actor's state which have the same property. The Stash trait’s implementation ofpreRestart will call unstashAll(), which is usually the desired behavior.

**Killing an Actor**

You can kill an actor by sending a Kill message. This will restart the actor through regular supervisor semantics.

Use it like this:

1. import akka.actor.PoisonPill;
2. import akka.actor.Kill;
3. victim.tell(Kill.getInstance(), null);

**Actors and exceptions**

It can happen that while a message is being processed by an actor, that some kind of exception is thrown, e.g. a database exception.

**What happens to the Message**

If an exception is thrown while a message is being processed (i.e. taken out of its mailbox and handed over to the current behavior), then this message will be lost. It is important to understand that it is not put back on the mailbox. So if you want to retry processing of a message, you need to deal with it yourself by catching the exception and retry your flow. Make sure that you put a bound on the number of retries since you don't want a system to livelock (so consuming a lot of cpu cycles without making progress). Another possibility would be to have a look at the *[PeekMailbox pattern](http://doc.akka.io/docs/akka/2.1.4/contrib/peek-mailbox.html" \l "mailbox-acking)*.

**What happens to the mailbox**

If an exception is thrown while a message is being processed, nothing happens to the mailbox. If the actor is restarted, the same mailbox will be there. So all messages on that mailbox will be there as well.

**What happens to the actor**

If code within an actor throws an exception, that actor is suspended and the supervision process is started (see [*Supervision and Monitoring*](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#supervision)). Depending on the supervisor’s decision the actor is resumed (as if nothing happened), restarted (wiping out its internal state and starting from scratch) or terminated.

**Initialization patterns**

The rich lifecycle hooks of Actors provide a useful toolkit to implement various initialization patterns. During the lifetime of an ActorRef, an actor can potentially go through several restarts, where the old instance is replaced by a fresh one, invisibly to the outside observer who only sees the ActorRef.

One may think about the new instances as "incarnations". Initialization might be necessary for every incarnation of an actor, but sometimes one needs initialization to happen only at the birth of the first instance when the ActorRef is created. The following sections provide patterns for different initialization needs.

**Initialization via constructor**

Using the constructor for initialization has various benefits. First of all, it makes it possible to use val fields to store any state that does not change during the life of the actor instance, making the implementation of the actor more robust. The constructor is invoked for every incarnation of the actor, therefore the internals of the actor can always assume that proper initialization happened. This is also the drawback of this approach, as there are cases when one would like to avoid reinitializing internals on restart. For example, it is often useful to preserve child actors across restarts. The following section provides a pattern for this case.

**Initialization via preStart**

The method preStart() of an actor is only called once directly during the initialization of the first instance, that is, at creation of its ActorRef. In the case of restarts, preStart() is called from postRestart(), therefore if not overridden, preStart() is called on every incarnation. However, overriding postRestart() one can disable this behavior, and ensure that there is only one call to preStart().

One useful usage of this pattern is to disable creation of new ActorRefs for children during restarts. This can be achieved by overriding preRestart():

1. @Override
2. public void preStart() {
3. // Initialize children here
4. }
6. // Overriding postRestart to disable the call to preStart()
7. // after restarts
8. @Override
9. public void postRestart(Throwable reason) {
10. }
12. // The default implementation of preRestart() stops all the children
13. // of the actor. To opt-out from stopping the children, we
14. // have to override preRestart()
15. @Override
16. public void preRestart(Throwable reason, Option<Object> message) {
17. // Keep the call to postStop(), but no stopping of children
18. postStop();
19. }

Please note, that the child actors are *still restarted*, but no new ActorRef is created. One can recursively apply the same principles for the children, ensuring that their preStart() method is called only at the creation of their refs.

For more information see [*What Restarting Means*](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#supervision-restart).

**Initialization via message passing**

There are cases when it is impossible to pass all the information needed for actor initialization in the constructor, for example in the presence of circular dependencies. In this case the actor should listen for an initialization message, and use become() or a finite state-machine state transition to encode the initialized and uninitialized states of the actor.

1. private String initializeMe = null;
3. @Override
4. public void onReceive(Object message) throws Exception {
5. if (message.equals("init")) {
6. initializeMe = "Up and running";
7. getContext().become(new Procedure<Object>() {
8. @Override
9. public void apply(Object message) throws Exception {
10. if (message.equals("U OK?"))
11. getSender().tell(initializeMe, getSelf());
12. }
13. });
14. }
15. }

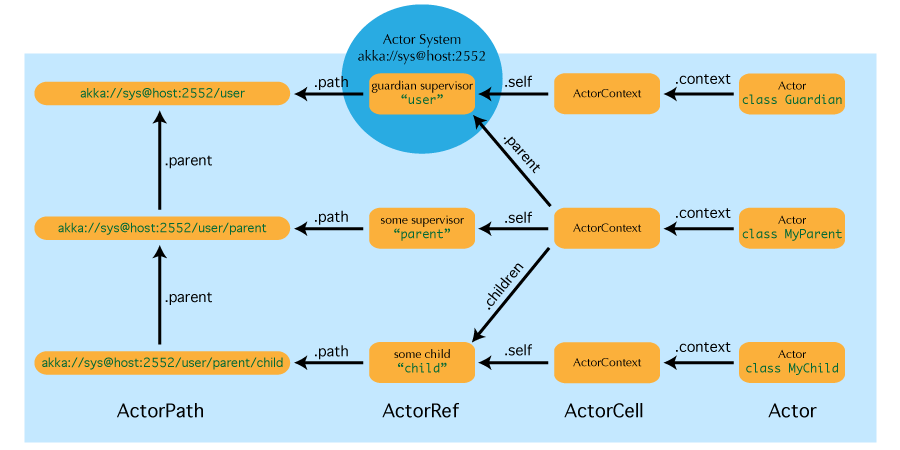
If the actor may receive messages before it has been initialized, a useful tool can be the Stash to save messages until the initialization finishes, and replaying them after the actor became initialized.

**Warning**

This pattern should be used with care, and applied only when none of the patterns above are applicable. One of the potential issues is that messages might be lost when sent to remote actors. Also, publishing an ActorRef in an uninitialized state might lead to the condition that it receives a user message before the initialization has been done.

Actor References, Paths and Addresses

This chapter describes how actors are identified and located within a possibly distributed actor system. It ties into the central idea that [Actor Systems](http://doc.akka.io/docs/akka/2.1.4/general/actor-systems.html#actor-systems) form intrinsic supervision hierarchies as well as that communication between actors is transparent with respect to their placement across multiple network nodes.



The above image displays the relationship between the most important entities within an actor system, please read on for the details.

## What is an Actor Reference?

An actor reference is a subtype of ActorRef, whose foremost purpose is to support sending messages to the actor it represents. Each actor has access to its canonical (local) reference through the self field; this reference is also included as sender reference by default for all messages sent to other actors. Conversely, during message processing the actor has access to a reference representing the sender of the current message through the sender field.

There are several different types of actor references that are supported depending on the configuration of the actor system:

* Purely local actor references are used by actor systems which are not configured to support networking functions. These actor references cannot ever be sent across a network connection while retaining their functionality.
* Local actor references when remoting is enabled are used by actor systems which support networking functions for those references which represent actors within the same JVM. In order to be recognizable also when sent to other network nodes, these references include protocol and remote addressing information.
* There is a subtype of local actor references which is used for routers (i.e. actors mixing in the Router trait). Its logical structure is the same as for the aforementioned local references, but sending a message to them dispatches to one of their children directly instead.
* Remote actor references represent actors which are reachable using remote communication, i.e. sending messages to them will serialize the messages transparently and send them to the other JVM.
* There are several special types of actor references which behave like local actor references for all practical purposes:
  + PromiseActorRef is the special representation of a Promise for the purpose of being completed by the response from an actor; it is created by the ActorRef.ask invocation.
  + DeadLetterActorRef is the default implementation of the dead letters service, where all messages are re-routed whose routees are shut down or non-existent.
  + EmptyLocalActorRef is what is returned when looking up a non-existing local actor path: it is equivalent to aDeadLetterActorRef, but it retains its path so that it can be sent over the network and compared to other existing actor refs for that path, some of which might have been obtained before the actor stopped existing.
* And then there are some one-off internal implementations which you should never really see:
  + There is an actor reference which does not represent an actor but acts only as a pseudo-supervisor for the root guardian, we call it “the one who walks the bubbles of space-time”.
  + The first logging service started before actually firing up actor creation facilities is a fake actor reference which accepts log events and prints them directly to standard output; it is Logging.StandardOutLogger.
* **(Future Extension)** Cluster actor references represent clustered actor services which may be replicated, migrated or load-balanced across multiple cluster nodes. As such they are virtual names which the cluster service translates into local or remote actor references as appropriate.

## What is an Actor Path?

Since actors are created in a strictly hierarchical fashion, there exists a unique sequence of actor names given by recursively following the supervision links between child and parent down towards the root of the actor system. This sequence can be seen as enclosing folders in a file system, hence we adopted the name “path” to refer to it. As in some real file-systems there also are “symbolic links”, i.e. one actor may be reachable using more than one path, where all but one involve some translation which decouples part of the path from the actor’s actual supervision ancestor line; these specialities are described in the sub-sections to follow.

An actor path consists of an anchor, which identifies the actor system, followed by the concatenation of the path elements, from root guardian to the designated actor; the path elements are the names of the traversed actors and are separated by slashes.

### Actor Path Anchors

Each actor path has an address component, describing the protocol and location by which the corresponding actor is reachable, followed by the names of the actors in the hierarchy from the root up. Examples are:

1. "akka://my-sys/user/service-a/worker1" // purely local
2. "akka://my-sys@host.example.com:5678/user/service-b" // local or remote
3. "cluster://my-cluster/service-c" // clustered (Future Extension)

Here, akka is the default remote protocol for the 2.0 release, and others are pluggable. The interpretation of the host & port part (i.e. serv.example.com:5678 in the example) depends on the transport mechanism used, but it must abide by the URI structural rules.

### Logical Actor Paths

The unique path obtained by following the parental supervision links towards the root guardian is called the logical actor path. This path matches exactly the creation ancestry of an actor, so it is completely deterministic as soon as the actor system’s remoting configuration (and with it the address component of the path) is set.

### Physical Actor Paths

While the logical actor path describes the functional location within one actor system, configuration-based remote deployment means that an actor may be created on a different network host than its parent, i.e. within a different actor system. In this case, following the actor path from the root guardian up entails traversing the network, which is a costly operation. Therefore, each actor also has a physical path, starting at the root guardian of the actor system where the actual actor object resides. Using this path as sender reference when querying other actors will let them reply directly to this actor, minimizing delays incurred by routing.

One important aspect is that a physical actor path never spans multiple actor systems or JVMs. This means that the logical path (supervision hierarchy) and the physical path (actor deployment) of an actor may diverge if one of its ancestors is remotely supervised.

### Virtual Actor Paths ****(Future Extension)****

In order to be able to replicate and migrate actors across a cluster of Akka nodes, another level of indirection has to be introduced. The cluster component therefore provides a translation from virtual paths to physical paths which may change in reaction to node failures, cluster rebalancing, etc.

This area is still under active development, expect updates in this section for the Akka release code named Rollins .

## How are Actor References obtained?

There are two general categories to how actor references may be obtained: by creating actors or by looking them up, where the latter functionality comes in the two flavours of creating actor references from concrete actor paths and querying the logical actor hierarchy.

While local and remote actor references and their paths work in the same way concerning the facilities mentioned below, the exact semantics of clustered actor references and their paths—while certainly as similar as possible—may differ in certain aspects, owing to the virtual nature of those paths. Expect updates for the Akka release code named Rollins.

### Creating Actors

An actor system is typically started by creating actors beneath the guardian actor using theActorSystem.actorOf method and then using ActorContext.actorOf from within the created actors to spawn the actor tree. These methods return a reference to the newly created actor. Each actor has direct access (through its ActorContext) to references for its parent, itself and its children. These references may be sent within messages to other actors, enabling those to reply directly.

### Looking up Actors by Concrete Path

In addition, actor references may be looked up using the ActorSystem.actorFor method, which returns a local or remote actor reference. The reference can be reused for communicating with said actor during the whole lifetime of the actor. In the case of a local actor reference, the named actor needs to exist before the lookup, or else the acquired reference will be an EmptyLocalActorRef. This will be true even if an actor with that exact path is created after acquiring the actor reference. For remote actor references the behaviour is different and sending messages to such a reference will under the hood look up the actor by path on the remote system for every message send.

#### Absolute vs. Relative Paths

In addition to ActorSystem.actorFor there is also ActorContext.actorFor, which is available inside any actor as context.actorFor. This yields an actor reference much like its twin on ActorSystem, but instead of looking up the path starting from the root of the actor tree it starts out on the current actor. Path elements consisting of two dots ("..") may be used to access the parent actor. You can for example send a message to a specific sibling:

1. context.actorFor("../brother") ! msg

Absolute paths may of course also be looked up on context in the usual way, i.e.

1. context.actorFor("/user/serviceA") ! msg

will work as expected.

### Querying the Logical Actor Hierarchy

Since the actor system forms a file-system like hierarchy, matching on paths is possible in the same way as supported by Unix shells: you may replace (parts of) path element names with wildcards («\*» and «?») to formulate a selection which may match zero or more actual actors. Because the result is not a single actor reference, it has a different type ActorSelection and does not support the full set of operations an ActorRefdoes. Selections may be formulated using the ActorSystem.actorSelection andActorContext.actorSelection methods and do support sending messages:

1. context.actorSelection("../\*") ! msg

will send msg to all siblings including the current actor. As for references obtained using actorFor, a traversal of the supervision hierarchy is done in order to perform the message send. As the exact set of actors which match a selection may change even while a message is making its way to the recipients, it is not possible to watch a selection for liveliness changes. In order to do that, resolve the uncertainty by sending a request and gathering all answers, extracting the sender references, and then watch all discovered concrete actors. This scheme of resolving a selection may be improved upon in a future release.

### Summary: actorOf vs. actorFor

**Note**

What the above sections described in some detail can be summarized and memorized easily as follows:

* actorOf only ever creates a new actor, and it creates it as a direct child of the context on which this method is invoked (which may be any actor or actor system).
* actorFor only ever looks up an existing actor, i.e. does not create one.

## Reusing Actor Paths

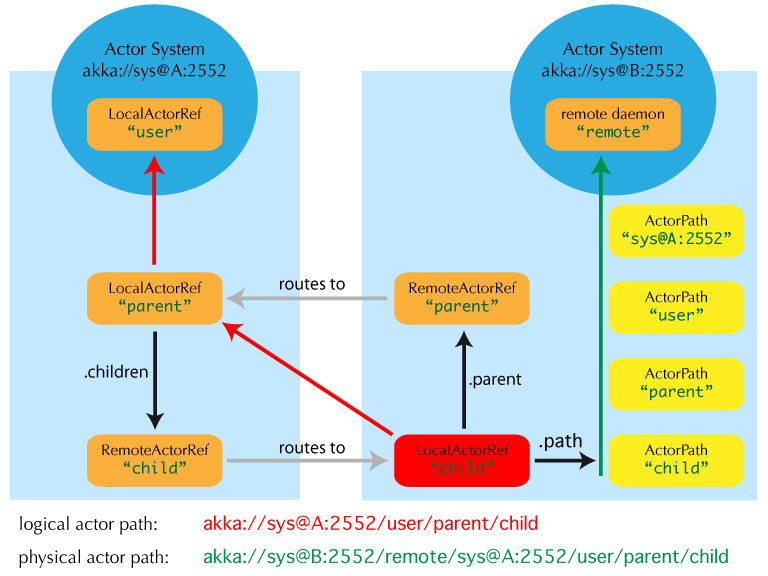
When an actor is terminated, its path will point to the dead letter mailbox, DeathWatch will publish its final transition and in general it is not expected to come back to life again (since the actor life cycle does not allow this). While it is possible to create an actor at a later time with an identical path—simply due to it being impossible to enforce the opposite without keeping the set of all actors ever created available—this is not good practice: remote actor references which “died” suddenly start to work again, but without any guarantee of ordering between this transition and any other event, hence the new inhabitant of the path may receive messages which were destined for the previous tenant.

It may be the right thing to do in very specific circumstances, but make sure to confine the handling of this precisely to the actor’s supervisor, because that is the only actor which can reliably detect proper deregistration of the name, before which creation of the new child will fail.

It may also be required during testing, when the test subject depends on being instantiated at a specific path. In that case it is best to mock its supervisor so that it will forward the Terminated message to the appropriate point in the test procedure, enabling the latter to await proper deregistration of the name.

## The Interplay with Remote Deployment

When an actor creates a child, the actor system’s deployer will decide whether the new actor resides in the same JVM or on another node. In the second case, creation of the actor will be triggered via a network connection to happen in a different JVM and consequently within a different actor system. The remote system will place the new actor below a special path reserved for this purpose and the supervisor of the new actor will be a remote actor reference (representing that actor which triggered its creation). In this case,context.parent (the supervisor reference) and context.path.parent (the parent node in the actor’s path) do not represent the same actor. However, looking up the child’s name within the supervisor will find it on the remote node, preserving logical structure e.g. when sending to an unresolved actor reference.



## The Interplay with Clustering ****(Future Extension)****

This section is subject to change!

When creating a scaled-out actor subtree, a cluster name is created for a routed actor reference, where sending to this reference will send to one (or more) of the actual actors created in the cluster. In order for those actors to be able to query other actors while processing their messages, their sender reference must be unique for each of the replicas, which means that physical paths will be used as self references for these instances. In the case of replication for achieving fault-tolerance the opposite is required: the self reference will be a virtual (cluster) path so that in case of migration or fail-over communication is resumed with the fresh instance.

## What is the Address part used for?

When sending an actor reference across the network, it is represented by its path. Hence, the path must fully encode all information necessary to send messages to the underlying actor. This is achieved by encoding protocol, host and port in the address part of the path string. When an actor system receives an actor path from a remote node, it checks whether that path’s address matches the address of this actor system, in which case it will be resolved to the actor’s local reference. Otherwise, it will be represented by a remote actor reference.

## Top-Level Scopes for Actor Paths

At the root of the path hierarchy resides the root guardian above which all other actors are found; its name is"/". The next level consists of the following:

* "/user" is the guardian actor for all user-created top-level actors; actors created using ActorSystem.actorOf are found below this one.
* "/system" is the guardian actor for all system-created top-level actors, e.g. logging listeners or actors automatically deployed by configuration at the start of the actor system.
* "/deadLetters" is the dead letter actor, which is where all messages sent to stopped or non-existing actors are re-routed (on a best-effort basis: messages may be lost even within the local JVM).
* "/temp" is the guardian for all short-lived system-created actors, e.g. those which are used in the implementation of ActorRef.ask.
* "/remote" is an artificial path below which all actors reside whose supervisors are remote actor references

The need to structure the name space for actors like this arises from a central and very simple design goal: everything in the hierarchy is an actor, and all actors function in the same way. Hence you can not only look up the actors you created, you can also look up the system guardian and send it a message (which it will dutifully discard in this case). This powerful principle means that there are no quirks to remember, it makes the whole system more uniform and consistent.

If you want to read more about the top-level structure of an actor system, have a look at [The Top-Level Supervisors](http://doc.akka.io/docs/akka/2.1.4/general/supervision.html#toplevel-supervisors).