Assignment
Transactor

To get started, download the reactive-protocols.zip handout archive file and extract it somewhere on your machine.

In this exercise we implement an Actor guarding a single value and offering transactional access to read or modify it—a so-called *Transactor*. We simplify this task by first creating a Behavior decorator for *selective receive*, programming the Transactor in Erlang style.

Selective Receive

As discussed in the video lecture on supervision in Akka Typed, Behaviors can be decorated or embedded in additional functionality. In this first part of the exercise we create a decorator that allows the embedded behavior to defer processing a message by returning `Behaviors.unhandled`; the message will then be put into a `StashBuffer` and retried upon each behavior change until it is finally accepted.

The interface of the decorator is a single function:

```
object SelectiveReceive {
    def apply[T](bufferSize: Int, initialBehavior: Behavior[T]):
Behavior[T] = ???
}
```

The `bufferSize` argument determines the capacity of the internal `StashBuffer`; when more unhandled messages are outstanding than fit into this buffer, the decorator shall raise a `StashOverflowException`. A buffer size of zero is legal and disallows buffering, essentially requiring all messages to be handled upon first arrival.

The decorator shall retry all outstanding messages after each accepted message, where retries shall occur in the order of message reception; after the first accepted retry, all remaining outstanding messages shall be retried from the head of the buffer, i.e. the retry always starts from the head of the buffer.

Transactor

A Transactor is a single-valued cell that allows transactions to be performed on the contained value. We implement it as an Actor that accepts the following protocol:

```
object Transactor {
    sealed trait PrivateCommand[T] extends Product with
Serializable
    final case class Committed[T](session: ActorRef[Session[T]],
value: T) extends PrivateCommand[T]
    final case class RolledBack[T](session: ActorRef[Session[T]])
extends PrivateCommand[T]

    sealed trait Command[T] extends PrivateCommand[T]
    final case class Begin[T](replyTo:
ActorRef[ActorRef[Session[T]]]) extends Command[T]

    sealed trait Session[T] extends Product with Serializable
    final case class Extract[T, U](f: T => U, replyTo: ActorRef[U])
extends Session[T]
    final case class Modify[T, U](f: T => T, id: Long, reply: U,
replyTo: ActorRef[U]) extends Session[T]
    final case class Commit[T, U](reply: U, replyTo: ActorRef[U])
extends Session[T]
    final case class Rollback[T]() extends Session[T]

    def apply[T](value: T, sessionTimeout: FiniteDuration):
Behavior[Command[T]] = ???
}
```

(extending `Product with Serializable` is only a hint to the type checker to improve error messages since all case classes extend these traits)

The protocol includes private messages that are not exposed to the general public. These model the protocol between the Transactor itself and the child actor it creates for each session, see below.

Behavior

The Transactor starts out with the given initial value. Other actors can interact with this value by first establishing an exclusive session and then exchanging messages with that session's `ActorRef`. The session commands are:

- `Extract`: apply the given projector function to the current Transactor value (possibly modified by the current session) and return the result to the given `ActorRef`; if the projector function throws an exception the session is terminated and all its modifications are rolled back
- `Modify`: calculate a new value for the Transactor by applying the given function to the current value (possibly modified by the current session already) and return the given reply value to the given replyTo `ActorRef`; if the function throws an exception the session is terminated and all its modifications are rolled back; the `id` argument serves as a deduplication identifier: sending the a `Modify` command with an `id` previously used within the current session will not apply the modification function but send the reply immediately
- `Commit`: terminate the current session, committing the performed modifications and thus making the modified value available to the next session; the given reply is sent to the given `ActorRef` as confirmation
- `Rollback`: terminate the current session rolling back all modifications, i.e. the next session will see the same value that this session saw when it started

The `sessionTimeout` parameter sets an upper limit on the duration of each single session; when an active session takes longer than this timespan then it will automatically be rolled back and the Transactor becomes available again for the next session.

Implementation

We implement the Transactor as one top-level actor with selective receive that handles incoming `Begin` messages, spawning one child actor for each session; new `Begin` messages are deferred while a session is ongoing. The depth of the selective receive buffer shall be 30 messages.

The Transactor uses DeathWatch to supervise each session, rolling back its effects upon failure. Upon session timeout the child actor for the active session is terminated.

Upon session `Commit`, the session's child actor sends a `Committed` message to the Transactor, informing it about the new value to be guarded and releasing the session such that the next session may be accepted.

Hints

- The `Committed` and `RolledBack` messages contain the session's `ActorRef` to allow the Transactor to safely ignore the messages pertaining to an old session—recall that all actor messaging and also termination are asynchronous.
- Modifications during an ongoing session are not communicated to the Transactor, only the final value is transmitted upon `Commit`; this way the Transactor is guaranteed to never store an intermediate value and thereby violate transactional behavior.

Submission

10 points possible (graded)

Run the packageSubmission command and upload the submission.jar file here.

Choose Files No file chosen

Submit