Chapter 2

Background and Related Work

2.1 The Actor Programming Model

The Actor Programming Model is first proposed by Hewitt et al. [1973] for the purpose of constructing concurrent systems. In the model, a concurrent system consists of actors which are primitive computational components. Actors communicate with each other by sending messages. Each actor independently reacts to messages it receives.

The Actor model given in [Hewitt et al., 1973] does not specify its formal semantics and hence does not suggest implementation strategies neither. An operational semantics of the Actor model is developed by Gerif [Grief, 1975]. Baker and Hewitt [1977] later defines a set of axiomatic laws for Actor systems. Other semantics of the Actor model includes the denotational semantics given by Clinger [1981] and the transition-based semantic model by Agha [1985]. Meanwhile, the Actor model has been implemented in Act 1 [Lieberman, 1981], a prototype programming language. The model influences designs of Concurrency Oriented Programming Languages (COPLs), especially the Erlang programming language [Armstrong, 2007b], which has been used in enterprise-level applications since it was developed in 1986.

A recent trend is adding actor libraries to full-fledged popular programming languages that do not have actors built-in. Some of the recent actor libraries are: JActor [JActor Consulting Ltd, 2013] for the JAVA language, Scala Actor [Haller and Odersky, 2006, 2007] for Scala, Akka [Typesafe Inc. (b), 2012] for Java and Scala, and CloudHaskell [Epstein et al., 2011] for Haskell.

2.2 The Supervision Principle

The core idea of the supervision principle is that actors should be monitored and restarted when necessary by their supervisors in order to improve the availability of a software system. The supervision principle is first proposed in the Erlang OTP library [Ericsson AB., 2013c] and is adopted by the Akka library [Typesafe Inc. (b), 2012].

A supervision tree in Erlang consists of two types of actors: workers and supervisors. A worker implements part of the business logic and reacts to request messages. A supervisor is responsible for initializing and monitoring its children, which are workers or supervisors for other actors, and restarting its children when necessary. The behaviour of a supervisor is defined by its *supervision strategy*.

The Akka library makes supervision obligatory. In Akka, every user-created actor is either a child of the system guidance actor or a child of another user-created actor. Therefore, every Akka actor is potentially the supervisor of some other actors. Different from the Erlang system, an Akka actor can be both a worker and a supervisor.

2.3 The Erlang Programming Language

Erlang [Armstrong, 2007a,b] is a dynamically typed functional programming language originally designed at the Ericsson Computer Science Laboratory for implementing telephony applications [Armstrong, 2007a]. After using the Erlang language for in-house applications for ten years, when Erlang was released as open source in 1998, Erlang developers summarised five design principles shipped with the Erlang/OTP library, which stands for Erlang Open Telecom Platform [Armstrong, 2007a; Ericsson AB., 2013c].

In enterprise-level applications, Erlang typically collaborates with other languages to provide fault-tolerant support for distributed real-time applications. One of the early OTP applications, Ericsson's AXD 301 switch, is reported to have achieved nine "9"s availability, that is 99.9999999% of uptime, during the nine months experiment [Armstrong, 2002]. Up to the present, Erlang has been widely used in database systems (e.g. Mnesia, Riak, and Amazon SimpleDB) and messaging services (e.g. RabbitMQ and WhatsApp).

This section gives a brief introduction to the Erlang programming language and OTP design principles, based on related material in [Armstrong, 2007b] and [Ericsson AB., 2013a,b,c].

2.3.1 Actor Programming in Erlang

(This section summarises material from [Armstrong, 2007b, Chapter 8] and [Ericsson AB., 2013b, Chapter 3])

An Erlang application consists of one or more module files, each of which defines a set of functions. The notion of the Erlang *process* minimizes the gap between sequential programming and concurrent programming. In Erlang, a process is a thread of function execution. It can receive messages of any type via its process identifier (pid). Defining an Actor in Erlang is as simple as providing a `receive` block for the body of the function spawned in a process.

2.3.1.1 Processes Creation

A *process* in Erlang is a thread of function execution. A process is created by calling the `spawn` method. Figure 2.1 gives the API of the `spawn` method [Ericsson AB., 2013a]. Calling `spawn(Module, Function, Args)` creates a process that executes the function `Module:Function(Args)`, where `Args` is a list of arguments. The `spawn` method returns a process identifier (pid) of the created process, which terminates when the execution of the function completes.

```
spawn(Module, Function, Args) -> pid()

  Module = Function = atom()
  Args = [Arg1,...,ArgN]
  ArgI = term()
```

Figure 2.1: Erlang API: spawn

To demonstrate the creation and usage of Erlang processes, Figure 2.2 shows the `echo` example modified from [Ericsson AB., 2013b, the `tut14` module] and its test result. As the terminal output shows, the `say_something` function prints out its first argument for the number of times specified by its second argument. What is more interesting is the result of `echo:start()`, which spawns two processes. The result shows that the execution of the two processes and the main thread, which returns a pid of the last `spawn` (i.e. `<0.41.0>`), are in parallel. As a consequence, the program prints out "hello", "goodbye", and the pid in a non-deterministic order.


```
-module(echo).

-export([start/0, say_something/2]).

say_something(_, 0) ->
    done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(echo, say_something, [hello, 3]),
    spawn(echo, say_something, [goodbye, 3]).

%% Terminal Output:
%% 1> c(echo).
%% {ok,echo}
%% 2> echo:say_something(hello, 3).
%% hello
%% hello
%% hello
%% done
%% 3> echo:start().
%% hello
%% goodbye
%% <0.41.0>
%% hello
%% goodbye
%% hello
%% goodbye
```

Figure 2.2: Erlang Example: An Echo Process

2.3.1.2 Message Passing Style Concurrency

In the `echo` example, the two processes are executed independently. To be an actor, an Erlang process shall be able to receive messages from others and reacts to messages.

In Erlang, users can send a message to a process via its pid using the `!` primitive. For example, the code

```
Pid ! Msg
```

will send the message `Msg` to the process whose pid is `Pid`. Message sending is an asynchronous operation and its evaluation result is the evaluated value of the sent message.

Messages sent to a process is queued in the mailbox of the recipient. To handle a received message, a process provides a `receive` block with the following syntax:

```
receive
  Message1 [when Guard1] ->
    Action1 ;
  Message2 [when Guard2] ->
    Action2 ;
  ...
  MessageN [when GuardN] ->
    ActionN
end
```

Figure 2.3: Erlang receive block

In the Erlang code pattern given in Figure 2.3, `receive` and `end` are primitives that denote the scope of the receive block. A receive block defines a set of guarded message patterns to which the current processed message will be matched in order. If the current message matches a pattern, the corresponding action will be evaluated. If the current message does not match any pattern, it will be saved in the mailbox and the next message in the mailbox will be processed. When reaching a receive block, the evaluation of the process will be suspended until at least one message in the mailbox matches one of the guarded patterns.

Generally speaking, the order in which messages appear in the mailbox is not necessarily the same as the order those messages were sent because messages may be concurrently sent from parallel threads or distributed nodes. Nevertheless, messages sent from the same sender to the same receiver is guaranteed to appear in the mailbox in the order they were sent, if both are delivered.

The `echo_actor` example, given in Figure 2.4, spawns two processes, both of which verbatim print their received messages. The `print` function terminates as soon as the first message is processed. On the contrary, `loop` is a recursive function that can always process new messages. Line 34 of Figure 2.4 confirms two properties of message sending in Erlang. Firstly, message sending is always a successful operation that returns the value of the sent message. At line 24, `P` is a pid pointing to a terminated process. Nevertheless, sending a message to `P` is still permitted. Secondly, message sending is an asynchronous operation. In this example, the evaluation result of line 23 is printed out after the evaluation

result of the `start` function (i.e. `hello4`), probably because it takes some time to match the message sent in line 23.

```
1  -module(echo_actor).
2
3  -export([start/0, loop/0, print/0]).
4
5  loop() ->
6      receive
7          Msg ->
8            io:format("loop: ~p~n", [Msg]),
9            loop()
10     end.
11
12 print() ->
13     receive
14         Msg ->
15           io:format("print: ~p~n", [Msg])
16     end.
17
18 start() ->
19     L = spawn(echo_actor, loop, []),
20     P = spawn(echo_actor, print, []),
21     L ! hello1,
22     P ! hello2,
23     L ! hello3,
24     P ! hello4.
25
26
27 %% Terminal Output:
28
29 %% 1> c(echo_actor).
30 %% {ok,echo_actor}
31 %% 2> echo_actor:start().
32 %% loop: hello1
33 %% print: hello2
34 %% hello4
35 %% loop: hello3
```

Figure 2.4: Erlang Example: An Echo Actor

2.3.1.3 An Erlang Actor with State

Erlang is a functional programming language where the value of a variable is immutable once assigned. On the other side, the result of a computation, for example, a search query, often depends on the value of some internal states.

Therefore, an Erlang actor needs to retain or update its internal states when it update its behaviour. One common method to define an Erlang actor with internal state is passing the state to the behaviour function.

The counter example defined in Figure 2.5 has one state variable, Val, which records the number of messages it has processed. The internal state is initialized to 0 when the actor is created at line 5. The value of the state is incremented each time when a message is processed (line 14 and line 17).

```
1  -module(counter).
2  -export([start/0, counter/1]).
3
4  start() ->
5      S = spawn(counter, counter, [0]),
6      S ! increment,
7      send_msgs(S, 3),
8      S.
9
10 counter(Val) ->
11     receive
12         increment ->
13           io:fwrite("increase counter to ~w~n", [Val+1]),
14           counter(Val+1);
15         Msg ->
16           io:fwrite("~w message(s) that has/have been processed ~n",
17               [Val+1]),
18           counter(Val+1)
19     end.
20
21 send_msgs(_, 0) -> true;
22 send_msgs(S, Count) ->
23     S ! "Hello",
24     send_msgs(S, Count-1).
25
26 %% Terminal Output:
27 %% 1> c(counter).
28 %% {ok,counter}
29 %% 2> counter:start().
30 %% increase counter to 1
31 %% <0.95.0>
32 %% 2 message(s) has/have been processed
33 %% 3 message(s) has/have been processed
34 %% 4 message(s) has/have been processed
```

Figure 2.5: Erlang Example: A Message Counter

Now I don't know. What would you do?

shall I compress this section?

2.3.2 Supervision in Erlang

(This section summarises material from [Ericsson AB., 2013c, Chapter 5])

Supervision is probably the most important concept in the OTP design principles [Ericsson AB., 2013c]. A supervision tree consists of workers and supervisors. Workers are processes which carry out actual computations while supervisors are processes which inspect a group of workers or sub-supervisors. Since both workers and supervisors are processes and they are organised in a tree structure, the term *child* is used to refer to any supervised process. The structure of a supervision tree may look like the one presented in Figure 2.6, where supervisors are represented by squares and workers are represented by circles. The example is cited from [Ericsson AB., 2013c, Section 1.1]. Restart strategies of each supervisor (Section 2.3.2.2), however, are removed from the figure since they are not related to the central ideas discussed at this moment.

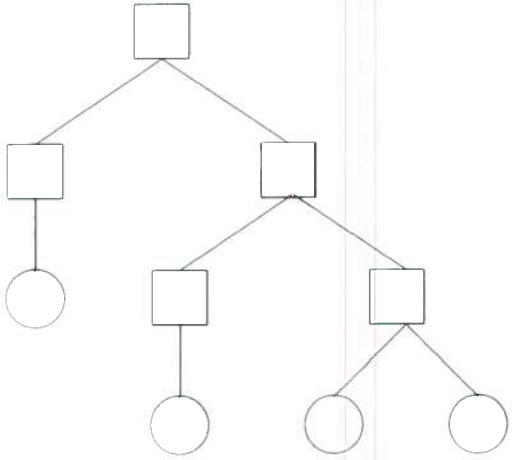

Figure 2.6: A Supervision Tree

2.3.2.1 An Erlang Supervision Example

We present the code for a simple Erlang supervisor in Figure 2.7. The core computation of the `supervision_demo` example is a problematic function `loop`, which eventually will try to compute the quotient of 10 divided by 0. As the test result shows, the problematic process has been restarted twice when it raises an error. The process is killed when it has failed for the third time within 60 seconds.

The short example implements the `supervisor` behaviour and specifies its supervision policy in its `init/1` method. The supervision policy reads as the following: spawn a worker process by calling `spawn(supervisor_demo, start, [Count])`; always restart the child when it fails if it does not fail more than twice within 60 seconds; if the child process fails more frequently than allowed, terminate it immediately. Alternative Erlang supervision strategies are explained in the following.

2.3.2.2 Supervision Strategy

In principle, a supervisor is accountable for starting, stopping and monitoring its child processes according to the policy specified in its `init/1` method, according to the API given in Figure 2.8 [Ericsson AB., 2013c]. A supervision strategy contains two parts: a restart strategy applies to all children and a list of child specifications for each child.

An Erlang supervisor employs one of the four restart strategies which specify its behaviour when one of its child fails. A supervisor with `one_for_one` strategy restart a child when it fails. A supervisor with `one_for_all` strategy restart all children when one of them fails. A supervisor with `rest_for_one` strategy restart the failed child and other children that started later than the failed child, according to their order in the list of child specifications. A supervisor with `simple_one_for_one` strategy is a `one_for_one` supervisor whose children are dynamically added instances of the same process.

A child specification contains 6 pieces of information [Ericsson AB., 2013c]: i) an internal name of the supervisor to identify the child. ii) the function call to start the child process. iii) whether the child process should be restarted after the termination of its siblings or itself. iv) how to terminate the child process. v) whether the child process is a worker or a supervisor. vi) a singleton list which specifies the name of the callback module.


```
-module(supervisor_demo).
-behaviour(supervisor).

-export([start/0, start/1, loop/1, init/1]).

start() ->
   supervisor:start_link(supervisor_demo, [2]).

start(Count) ->
  io:fwrite("Starting...~n"),
  Pid=spawn_link(supervisor_demo, loop, [Count]),
  {ok, Pid}.

init([Count]) ->
  {ok, {{one_for_one, 2, 60},
        [{supervisor_demo, {supervisor_demo, start, [Count]},
        permanent, brutal_kill, worker, [supervisor_demo]}]}}.

loop(Count) ->
  io:fwrite("~w / ~w is ~w ~n", [10, Count, 10/Count]),
  loop(Count-1).

%% 1> c(supervisor_demo).
%% {ok,supervisor_demo}
%% 2> supervisor_demo:start().
%% Starting...
%% 10 / 2 is 5.0
%% 10 / 1 is 10.0
%% <0.39.0>
%% Starting...
%% 10 / 2 is 5.0
%% 10 / 1 is 10.0
%% Starting...
%% 3>
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.40.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% 10 / 2 is 5.0
%% 3>
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.42.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% 10 / 1 is 10.0
%% =ERROR REPORT==== 14-Oct-2013::00:03:49 ===
%% Error in process <0.43.0> with exit value:
{badarith,[{supervisor_demo,loop,1,[{file,"supervisor_demo.erl"},{line,20}]}]}
%% ** exception error: shutdown
```

Figure 2.7: Erlang Example: Supervision Demo

```
Module:init(Args) -> Result

Args = term()
Result = {ok, {{RestartStrategy,MaxR,MaxT},[ChildSpec]}} | ignore
   RestartStrategy = strategy()
   MaxR = integer()>=0
   MaxT = integer()>0
   ChildSpec = child_spec()

child_spec() =
    {Id :: child_id(),
     StartFunc :: mfargs(),
     Restart :: restart(),
     Shutdown :: shutdown(),
     Type :: worker(),
     Modules :: modules()}

child_id() = term()

mfargs() =
    {M :: module(), F :: atom(), A :: [term()] | undefined}

restart() = permanent | transient | temporary

shutdown() = brutal_kill | timeout()

strategy() = one_for_all
           | one_for_one
           | rest_for_one
           | simple_one_for_one

worker() = worker | supervisor

modules() = [module()] | dynamic
```

Figure 2.8: Erlang API: supervision

No, shall I keep §2.3.3, 4 only?

2.3.3 Other OTP Design Principles

Based on 10 years of experience of using Erlang in Enterprise level applications, Erlang developers summarized 5 OTP design principles in 1999 to improve the reliability of Erlang applications [Ericsson AB., 2013c]: The Behaviour Principle, The Application Principle, The Release Principle, The Release Handling Principle, and The Supervision Principle. The Supervision Principle has been introduced in the previous section. This section describes the idea of the remaining 4 OTP design principles and the methodology of applying them in a JVM based environment, such as Java and Scala. The Supervision principle, which is the central topic of this thesis, has no direct correspondence in native Java and Scala programming.

2.3.3.1 The Behaviour Principle

A Behaviour in Erlang is similar to an interface, a trait, or an abstract class in the objected oriented programming. It implements common structures and patterns of process implementations. With the help of behaviours, Erlang code can be divided into a generic part, a behaviour module, and a specific part, a callback module. Most processes, including the supervisor in Section 2.3.2, is coded by implementing a set of pre-defined callback functions for one or more behaviours. Although ad-hoc code and programming structures may be more efficient, using consistent general interfaces make code more maintainable and reliable. Standard Erlang/OTP behaviours include:

- *gen_server* for constructing the server of a clientserver paradigm.
- *gen_fsm* for constructing finite state machines.
- *gen_event* for implementing event handling functionality.
- *supervisor* for implementing a supervisor in a supervision tree.

2.3.3.2 The Application Principle

A software system on the OTP platform is made of a group of components called applications. To define an application, users implements two callback functions of the `application` behaviour: `start/2` and `stop/1`. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process, registered as `application_controller`.

Distributed applications may be deployed on several distributed Erlang nodes. An Erlang distributed application will be restarted at another node when its current node goes down. A distributed application is controlled by both the application controller and the distributed application controller, registered as `dist_ac`, both of which are part of the *kernel* application. Two configuration parameters must be set before loading and launching a distributed application. First, possible nodes where the distributed application may run must be explicitly pointed. Second, all nodes configured in the last step will be sent a copy of the same configuration which include three parameters: the time for other nodes to start, nodes that *must* be started in a given time, and nodes that *may* be started in a given time.

2.3.3.3 The Release Principle and The Release Handling Principle

A complete Erlang system consists of one or more applications, packaged in a release resource file. Different version of a release can be upgraded or downgraded at run-time dynamically by calling APIs in the `release_handler` module in the SASL (System Architecture Support Libraries) application. Hot swapping on an entire release application is a distinct feature of Erlang/OTP, which aims at design and running non-stop applications.

2.3.3.4 Appling OTP Design Principles in Java and Scala

To sum, we make an analogy between Erlang/OTP design principles and common practices in Java and Scala programming, summarised in Table 2.1.

OTP Design Principle	Java/Scala Analogy
Behaviour	defining an abstract class, an interface, or a trait.
Application	defining an abstract class that has two abstract methods: start and stop
Release	packaging related application classes
Release Handling	hot swapping support on key modules is required
Supervision	no direct correspondence

Table 2.1: Using OTP Design Principles in JAVA and Scala Programming

First, the notion of callback functions in Erlang/OTP is close to to the notion of abstract methods in Java and Scala. An OTP behaviour that only defines the signature of callback functions can be ported to Java and Scala as an interface. An OTP behaviour that implements some behaviour functions can be ported as an abstract class to *prevent* multiple inheritance, or a trait to *permit* multiple

inheritance. Since Java does not have the notion of trait, porting an Erlang/OTP module that implements multiple behaviours requires a certain amount of refactoring work.

Second, since the Erlang application module is just a special behaviour, we can define an equivalent interface `Application` which contains two abstract methods: `start` and `stop`. To mimic the dynamic type system of Erlang system, the `start` method may be declared as
`public static void start(String name, Object... arguments)` and as
`def start(name:String, arguments:Any*):Unit` in Java and Scala respectively.

Third, Erlang releases correspond to packages in Java and Scala but hot code swapping is not directly supported by JVM. During the development of the TAkka library, we noticed that hot code swapping on a key component can be mimicked by updating the reference to that component.

The final OTP design principle, Supervision, has no direct correspondence in Java and Scala programming practices. The next section introduces the Akka library which implements the supervision principle.

2.4 The Akka Library

Akka is the first library that makes supervision obligatory. The API of the Akka library [Typesafe Inc. (a), 2012; Typesafe Inc. (b), 2012] is similar to the Scala Actor library [Haller and Odersky, 2006, 2007], which borrows syntax from the Erlang languages [Armstrong, 2007b; Ericsson AB., 2013a]. Both Akka and Scala Actor are built in Scala, a typed language that merges features from Object-Oriented Programming and Functional Programming. This section gives a brief tutorial on Akka, based on related materials in the Akka Documentation [Typesafe Inc. (b), 2012].

2.4.1 Actor Programming in Akka

(This section summarises material from [Typesafe Inc. (b), 2012, Section 2.3 and 3.1])

Although many Akka designs have their origin in Erlang, the Akka Team at Typesafe Inc. devises a set of connected concepts that explains Actor programming in the Akka framework. This subsection begins with a short Akka example, followed by elaborate explanations of involved concepts.

```
1  package sample.akka
2
3  import akka.actor.{Actor, ActorRef, ActorSystem, Props}
4
5  class StringCounter extends Actor {
6    var counter = 0;
7    def receive = {
8      case m:String =>
9        counter = counter +1
10       println("received "+counter+" message(s):\n\t"+m)
11   }
12 }
13
14 class MessageHandler extends Actor {
15   def receive = {
16     case akka.actor.UnhandledMessage(message, sender, recipient) =>
17       println("unhandled message:"+message);
18   }
19 }
20
21 object StringCounterTest extends App {
22   val system = ActorSystem("StringCounterTest")
23   val counter = system.actorOf(Props[StringCounter], "counter")
24
25   val handler = system.actorOf(Props[MessageHandler]))
26   system.eventStream.subscribe(handler,classOf[akka.actor.UnhandledMessage]);
27   counter ! "Hello World"
28   counter ! 1
29   val counterRef =
         system.actorFor("akka://StringCounterTest/user/counter")
30   counterRef ! "Hello World Again"
31   counterRef ! 2
32 }
33
34
35 /*
36 Terminal output:
37 received 1 message(s):
38   Hello World
39 received 2 message(s):
40   Hello World Again
41 unhandled message:1
42 unhandled message:2
43 */
```

Figure 2.9: Akka Example: A String Counter

The code presented in Figure 2.9 defines and uses an actor which counts String messages it receives. An Akka actor implements its message handler by defining a `receive` method of type `PartialFunction[Any, Unit]`. In the `StringCounterTest` application, we create an Actor System (Section 2.4.1.1), initialise an actor (Section 2.4.1.2) inside the Actor System by passing a corresponding `Props` (Section 2.4.1.4), and send messages to the created actor via its actor references (Section 2.4.1.5). Unexpected messages to the `counter` actor (e.g. line 28 and 31) are handled by an instance of `MessageHandler`, a helper actor for the test application. Lastly, the order in which the four output messages are printed is non-deterministic, but "Hello World" is always printed before "Hello World Again" and "unhandled message:1" is always printed before "unhandled message:2". Explain why.

2.4.1.1 Actor System

In Akka, every actor is resident in an Actor System. An actor system organises related actors in a tree structure and provides services such as thread scheduling, network connection, and logging. One or several local and remote actor systems consist a complete application.

To create an actor system, users provide a name and an optional configuration to the `ActorSystem` constructor. For example, an actor system is created in Figure 2.9 by the following code.

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

In the above, an actor system of name StringCounterTest is created at the machine where the program runs. The above created actor system uses the default Akka system configuration which provides a simple logging service, a round-robin style message router, but does not support remote messages. Customized configuration can be encapsulated in a `Config` instance and passed to the `ActorSystem` constructor, or specified as part of the application configuration file. This short tutorial will not look into customized configurations, which have minor differences in different Akka versions, and are not related to our central topics.

2.4.1.2 The Actor Class

An Akka Actor has four groups of fields given in Figure 2.10: *i*) its *state*, *ii*) its *behaviour* functions, *iii*) an `ActorContext` instance encapsulating its contextual information, and *iv*) the *supervisor strategy* for its children. This subsection

explains the *state* and *behaviour* of actors, which are required when defining an Actor class. Overriding default *actor context* and *supervisor strategy* will be explained in later subsections.

```
trait Actor extends AnyRef
  type Receive = PartialFunction[Any, Unit]

  abstract def receive: Actor.Receive
  implicit final val self: ActorRef
  implicit val context: ActorContext
  def supervisorStrategy: SupervisorStrategy

  final def sender: ActorRef

  def preStart(): Unit
  def preRestart(reason: Throwable, message: Option[Any]): Unit
  def postRestart(reason: Throwable): Unit
  def postStop(): Unit}
```

Figure 2.10: Akka API: Actorr

An Akka actor may contain some mutable variables and immutable values that represent its *internal state*. Each Akka actor has an actor reference, `self`, to which messages can be sent to that actor. The value of `self` is initialised when the actor is created. Notice that `self` is declared as a value field (`val`), rather than a variable field (`var`), so that its value cannot be changed. In addition to *immutable states*, sometimes *mutable states* are also required. For example, Akka developers believe that the sender of the last message shall be recorded and easily fetched by calling the `sender` method. In the `StringCounter` example, we straightforwardly add a `counter` variable which is initialized to 0 and is incremented each time when a String message is processed.

There are two drawbacks of using mutable internal variables to represent states. Firstly, those variables will be reset each time when the actor is restarted, either due to a failure caused by itself or be enforced by its supervisor for other reasons. Secondly, mutable internal variables result in the difficulty of implementing a consistent cluster environment where actors may be replicated to increase reliability [Kuhn et al., 2012]. The alternatives of working with mutable states will be discussed in Section 3.10.

There are two kinds of `behaviour` functions of an actor. The first type of behaviour functions is a `receive` function which defines its action to incoming messages. The `receive` function is declared as an `abstract` function,

which must be implemented otherwise the class cannot be initialised. The second group of behaviour functions has four overridable functions which are triggered before the actor is started (`preStart`), before the actor is restarted (`preRestart`), after the actor is restarted (`postRestart`), and when the actor is permanently terminated (`postStop`). The default implementation of those four functions take no action when they are invoked.

Look closely at the `receive` function of the `StringCounter` actor in Figure 2.9, it actually has type `Function[String, Unit]` rather than the declared type `PartialFunction[Any, Unit]`. The definition of `StringCounter` is accepted by the Scala compiler because `PartialFunction` does not check the completeness of the input patterns. The behaviour of processing non-String messages, however, is undefined in the `receive` method.

2.4.1.3 Message Mailbox

An actor receives messages from other parts of the application. Arrived messages are queued in its sole mailbox to be processed. Different from the Erlang design (Section 2.3.1.2), the behaviour function of an Akka actor must be able to process the message it is given. If the message does not match any message pattern of the current behaviour, a failure arises.

Undefined messages are treated differently in different Akka versions. In versions prior to 2.0, an Akka actor raises an exception when it processes an undefined message. It means that sending an ill-typed message will cause a failure at the receiver side. In Akka 2.1, an undefined message is discarded by the actor and an `UnhandledMessage` event is pushed to the event stream of the actor system. The event stream may be subscribed by other actors who are interested in particular event messages. Line 24 of the String Counter example demonstrates how to subscribe a type of messages in the event stream of an actor system.

2.4.1.4 Actor Creation with Props

An instance of the `Props` class, which ~~perhaps~~ stands for "properties", specifies the configuration of creating an actor. A `Props` instance is immutable so that it can be consistently shared between threads and distributed nodes.

Figure 2.11 gives part of the APIs of the `Props` class and its companion object. The `Props` is defined as a *final class* so that users cannot define subclasses of it. Moreover, users are not encouraged to initialise a `Props` instance by directly using its constructor. Instead, a `Props` should be initialised by using one of the

apply methods supplied by the Props object. From the perspective of software design patterns, the Props object is a *Factory* for creating instances of the Props class.

```
package akka.actor
final case class Props(deploy: Deploy, clazz: Class[_],
                       args: Seq[Any]) extends Product with Serializable

object Props extends Serializable
  def apply[T <: Actor]()(implicit arg0: ClassManifest[T]): Props
  def apply(clazz: Class[_], args: Any*): Props
```

Figure 2.11: Akka API: Props

We have seen an example of creating a Props instance in Figure 2.9, that is:

```
Props[StringCounter]
```

which is short for

```
Props.apply[StringCounter]()(implicitly[ClassManifest[StringCounter]])
```

The API of the first Props.apply method is carefully designed to take the advantages of the Scala language. Firstly, the word apply can be omitted when it is used as a method name. Secondly, round brackets can be omitted when a method does not take any argument. Thirdly, implicit parameters are automatically provided if implicit values of the right types can be found in scope. As a result, in most cases, only the class name of an Actor is required when creating a Props of that actor.

Alternatively, calling the second apply method requires a *class object* and arguments sending to the class constructor. For example, the above Props can be alternatively created by the following code:

```
Props(classisOf[StringCounter])
```

In the above, the predefined function classOf[T] returns a class object for type T. More arguments can be sent to the constructor of StringCounter if there is one that requires more parameters. The signature of the constructor, including the number, types and order of its parameters, is verified at the run time. If no matched constructor is found when initializing the Props object, an IllegalArgumentException will arise.

Once an instance of Props is created, an actor can be created by passing that Props instance to the actorOf method of ActorSystem or ActorContext.

In Figure 2.9, we have seen that system.actorOf creates an actor directly supervised by the system guidance actor for all user-created actors (user). Calling context.actorOf creates an actor supervised by the actor represented by that context. Details of actor context and supervision will be given in Section 2.4.1.6 and Section 2.4.2 respectively.

2.4.1.5 Actor Reference and Actor Path

Actors collaborate by sending messages to each others via actor references of message receivers. An actor reference has type ActorRef, which provides a ! method to which messages are sent. For example, in the StringCounter example in Figure 2.9, counter is an actor reference to which the message "Hello world" is sent by the following code:

```
counter ! "Hello world"
```

which is the syntactic sugar for

```
counter.!("Hello world") .
```

```
abstract class ActorRef extends Comparable[ActorRef] with Serializable

abstract def path: ActorPath
def !(message: Any)(implicit sender: ActorRef = Actor.noSender): Unit
final def compareTo(other: ActorRef): Int
final def equals(that: Any): Boolean
def forward(message: Any)(implicit context: ActorContext): Unit
```

Figure 2.12: Akka API: Actor Reference

An actor path is a symbolic representation of the address where an actor can be located. Since actors forms a tree hierarchy in Akka, a unique address can be allocated for each actor by appending an actor name, which shall not be conflict with its siblings, to the address of its parent. Examples of akka addresses are:

```
"akka://mysystem/user/service/worker"                      //local
"akka.tcp://mysystem:example.com:1234/user/service/worker" //remote
"cluster://mycluster/service/worker"                       //cluster
```

The first address represents the path to a local actor. Inspired by the syntax of uniform resource identifier (URI), an actor address consists of a scheme name (akka), actor system name (e.g. mysystem), and names of actors from the guardian actor (user) to the respected actor (e.g. service, worker). The

second address represents the path to a remote actor. In addition to components of a local address, a remote address further specifies the communication protocol (`tcp` or `udp`), the IP address or domain name (e.g. `example.com`), and the port number (e.g. `1234`) used by the actor system to receive messages. The third address represents the desired format of a path to an actor in a cluster environment in a further Akka version. In the design, protocol, IP/domain name, and port number are omitted in the address of an actor which may transmit around the cluster or have multiple copies.

An actor path corresponds to an address where an actor can be identified. It can be initialized without the creation of an actor. Moreover, an actor path can be re-used by a new actor after the termination of an old actor. Two actor paths are considered equivalent as long as their symbolic representations are equivalent strings. On the contrary, an actor reference must correspond to an existing actor, either an alive actor located at the corresponding actor path, or the special `DeadLetter` actor which receives messages sent to terminated actors. Two actor references are equivalent if they correspond to the same actor path and the same actor. An restarted actor is considered as the same actor as the one before the restart because the life cycle of an actor is not visible to the users of `ActorRef`.

2.4.1.6 Actor Context

The `ActorContext` class has been mention a few times in previous sections. This section explains what the contextual information of an Akka actor includes, with a reference to the following APIs cited from [Typesafe Inc. (a), 2012].

The API in Figure 2.13 shows two groups of methods: those for interacting with other actors (line 3 to line 24), and those for controlling the behaviour of the represented actor (line 26 to line 30).

As mentioned in Section 2.4.1.4, calling the `context.actorOf` method creates a child actor supervised by the actor represented by that context. Every actor has a name distinguished from its siblings. If a user assigned name is conflict with the name of another existed actor, an `InvalidActorNameException` raises. If the user does not provide a name when creating an actor, a system generated name will be used instead. The return value of the `actorOf` method is an actor reference pointing to the created actor.

Once an actor is created, its actor reference can be obtained by inquiring on its actor path using the `actorFor` method. Since version 2.1, Akka encourages obtaining actor references via a new method `actorSelection`, whose return


```
1  package akka.actor
2  trait ActorContext
3    abstract def actorOf(props: Props, name: String): ActorRef
4    abstract def actorOf(props: Props): ActorRef
5
6    abstract def child(name: String): Option[ActorRef]
7    abstract def children: Iterable[ActorRef]
8    abstract def parent: ActorRef
9
10   abstract def props: Props
11   abstract def self: ActorRef
12   abstract def sender: ActorRef
13
14   implicit abstract def system: ActorSystem
15
16   def actorFor(path: Iterable[String]): ActorRef
17   def actorFor(path: String): ActorRef
18   def actorFor(path: ActorPath): ActorRef
19   def actorSelection(path: String): ActorSelection
20
21   abstract def watch(subject: ActorRef): ActorRef
22   abstract def unwatch(subject: ActorRef): ActorRef
23
24   abstract def stop(actor: ActorRef): Unit
25
26   abstract def become(behavior: Receive,
27                       discardOld: Boolean = true): Unit
28   abstract def unbecome(): Unit
29   abstract def receiveTimeout: Duration
30   abstract def setReceiveTimeout(timeout: Duration): Unit
```

Figure 2.13: Akka API: Actor Context

value broadcasts messages it receives to all actors in its subtrees. The `actorFor` method is deprecated in version 2.2. Code in this thesis still uses the deprecated `actorFor` method because, in most cases, our simple examples only need to send message to a particular actor.

Actor context is also used to fetch some states inside the actor. For example, the context of an actor records references to its parent and children, the props used to create that actor, actor references to itself and the sender of the last message, and the actor system where the actor is resident.

Ported from the Erlang design, using the `watch` method, an Akka actor can monitor the liveness of another actor, which is not necessarily its child. The liveness monitoring can be cancelled by calling the `unwatch` method. Another

method ported from Erlang is the `stop` method which sends a termination signal to an actor. Since supervision is obligatory in Akka and users are encouraged to managing the lifecycle of an actor either inside the actor or via its supervisor, we believe that those three methods are redundant in Akka. For all examples studied in this thesis, there is no client application that requires any of those three methods.

Finally, actor context manages two behaviours of the actor it represents. The first behaviour specified by the actor context is the timeout within which a new message shall be received or a `ReceiveTimeout` message is sent to the actor. The second behaviour managed by the actor context is the handler for incoming messages. The next subsection explains how to hot swap the message handler of an actor using the `become` and `unbecome` method.

2.4.1.7 Hot Swapping on Message Handler

In the `StringCounter` example given at the beginning of this section, a message handler is defined in the `receive` method. The `StringCounter` is a simple actor which only requires an initial message handler that never changes. In some other cases, it is required to update the message handler of an actor at runtime. For example, an online calculator may upgrade to a version that supports more types of calculation.

Message handlers of an Akka actor are kept in a stack of its `context`. A message handler is pushed to the stack when the `context.become` method is called; and is popped out from the stack when the `context.unbecome` method is called. The message handler of an actor is reset to the initial one, i.e. the `receive` method, when it is restarted.

To demonstrate hot swapping on the behaviour of Akka actors, Figure 2.14 defines a calculator. The calculator starts with a basic version that can only compute multiplication. When it receives an `Upgrade` command, it upgrades to an advanced version that can compute both multiplication and division. The advanced calculator downgrades to to the basic version when it receives a `Downgrade` command. For simplicity, the demo code does not consider the potential *division by zero* problem, an error that can be tolerated if the actor is properly supervised.


```
package sample.akka
import akka.actor._
case object Upgrade
case object Downgrade
case class Mul(m:Int, n:Int)
case class Div(m:Int, n:Int)
class CalculatorServer extends Actor {
  import context._
  def receive = simpleCalculator
  def simpleCalculator:Receive = {
    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
    case Upgrade =>
      println("Upgrade")
      become(advancedCalculator, discardOld=false)
    case op =>    println("Unrecognised operation: "+op)
  }
  def advancedCalculator:Receive = {
    case Mul(m:Int, n:Int) =>  println(m +" * "+ n +" = "+ (m*n))
    case Div(m:Int, n:Int) =>  println(m +" / "+ n +" = "+ (m/n))
    case Downgrade =>
      println("Downgrade")
      unbecome();
    case op =>    println("Unrecognised operation: "+op)
  }
}
object CalculatorUpgrade extends App {
    val system = ActorSystem("CalculatorSystem")
    val calculator:ActorRef = system.actorOf(Props[CalculatorServer],
        "calculator")
   calculator ! Mul(5, 1)
   calculator ! Div(10, 1)
   calculator ! Upgrade
   calculator ! Mul(5, 2)
   calculator ! Div(10, 2)
   calculator ! Downgrade
   calculator ! Mul(5, 3)
   calculator ! Div(10, 3)
}
/* Terminal output:
5 * 1 = 5
Unrecognised operation: Div(10,1)
Upgrade
5 * 2 = 10
10 / 2 = 5
Downgrade
5 * 3 = 15
Unrecognised operation: Div(10,3)
*/
```

Figure 2.14: Akka Behaviour Swap Example

2.4.2 Supervision in Akka

(This section summarises material from [Typesafe Inc. (b), 2012, Section 2.4 and 3.4])

A distinguishing feature of the Akka library is making supervision obligatory by restricting the way of creating actors. Recall that every user-created actor is initialised in one of the two ways: using the `system.actorOf` method so that it is a child of the system guardian actor; or using the context.actorOf method so that it is a child of another user-created actor. Therefore, all user-created actors in an actor system, together with the guardian actor of that actor system, form a tree structure. Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance. This section summarises concepts in the Akka supervision tree.

2.4.2.1 Children

Every actor in Akka is a supervisor for a list of other actors. An actor creates a new child by calling `context.actorOf` and removes a child by calling `context.stop(child)`, where `child` is an actor reference.

2.4.2.2 Supervisor Strategy

The Akka library implements two supervisor strategies: `OneForOne` and `AllForOne`. The `OneForOne` supervisor strategy corresponds to the `one_for_one` supervision strategy in OTP, which restart a child when it fails. The `AllForOne` supervisor strategy corresponds to the `one_for_all` supervision strategy in OTP, which restart all children when any of them fails. The `rest_for_all` supervision strategy in OTP is not implemented in Akka because Akka actor does not specify the order of children. Simulating the `rest_for_all` strategy in Akka requires ad-hoc implementation that groups related children and defines special messages to trigger actor termination. It is not clear whether the lack of the `rest_for_one` strategy will result in difficulties when rewriting Erlang applications in Akka.

Figure 2.15 gives API of Akk supervisor strategies. As in OTP, for each supervisor strategy, users can specify the maximum number of restarts permitted for its children within a period. The default supervisor strategy in Akka is `OneForOne` that permits unlimited restarts.

As shown in the API, an Akka supervisor strategy can choose different reactions for different reasons of child failures in its `decider` parameter. Recall that `Throwable` is the superclass of `Error` and `Exception` in Scala and Java.


```
package akka.actor
abstract class SupervisorStrategy
case class OneForOne(restart:Int, time:Duration)(decider: Throwable =>
  Directive) extends SupervisorStrategy with Product with Serializable
case class OneForAll(restart:Int, time:Duration)(decider: Throwable =>
  Directive) extends SupervisorStrategy with Product with Serializable

sealed trait Directive extends AnyRef
object Escalate extends Directive with Product with Serializable
object Restart extends Directive with Product with Serializable
object Resume extends Directive with Product with Serializable
object Stop extends Directive with Product with Serializable
```

Figure 2.15: Akka API: Supervisor Strategies

Therefore, users can pattern match on possible types and values of `Throwable` in the `decider` function. In other words, when the failure of a child is passed to the `decider` function of the supervisor, it is matched to a pattern that reacts to that failure.

The `decider` function contains user-specified computations and returns a value of `Directive` that denotes the standard recovery process implemented by the Akka library developers. The `Directive` trait is an enumerated type that has four possible values: the `Escalate` action which throws the exception to the supervisor of the supervisor, the `Restart` action which replaces the failed child with a new one, the `Resume` action which asks the child to process the message again, and the `Stop` action which terminates the failed actor permanently.

2.4.3 Case Study: A Fault-Tolerant Calculator

Figure 2.16 defines a simple calculator which supports multiplication and division. The simple calculator does not consider the problematic case of dividing a number by 0, in which case an `ArithmeticException` will raise. We then define a safe calculator as the supervisor of the simple calculator. The safe calculator delegates calculation tasks to the simple calculator and restart the simple calculator when an `ArithmeticException` raises. The supervisor strategy of the safe calculator also specifies the maximum failures its child may have within a time range. If the child fails more frequently than the allowed frequency, the safe calculator will be stopped, and its failure will be reported to its supervisor, the system guardian actor in this example. The terminal output shows that the simple calculator is restarted before the third and fifth message is delivered.

```
case class Multiplication(m:Int, n:Int)
case class Division(m:Int, n:Int)

class Calculator extends Actor {
  def receive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}

class SafeCalculator extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 2, withinTimeRange = 1 minute) {
      case _: ArithmeticException =>
        println("ArithmeticException Raised to: "+self)
        Restart
    }
  val child:ActorRef = context.actorOf(Props[Calculator], "child")
  def receive = {
    case m => child ! m
  }
}

  val system = ActorSystem("MySystem")
  val actorRef:ActorRef = system.actorOf(Props[SafeCalculator],
"safecalculator")

  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 2)
  calculator ! Division(10, 0)
  calculator ! Multiplication(3, 3)

/*
Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
10 / 5 = 2
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
3 * 2 = 6
ArithmeticException Raised to:
    Actor[akka://MySystem/user/safecalculator]
java.lang.ArithmeticException: / by zero
*/
```

Figure 2.16: Supervised Calculator

The last message is not processed because the both calculators are terminated because the simple calculator fails more frequently than allowed.

2.5 The Scala Type System

I rewrote the whole section.

One of the key design principles of the TAkka library, described in the subsequent Chapters, is using type checking to detect some errors at the earliest opportunity. Since both TAkka and Akka are built using the Scala programming language [Odersky et al., 2004; Odersky., 2013], this section summarises key features of the Scala type system that benefit the implementation of the TAkka library.

2.5.1 Parameterized Types

A *parameterized type* $\mathtt{T}[\mathtt{U}_1, \ldots, \mathtt{U}_n]$ consists of a type constructor T and a positive number of type parameters $\mathtt{U}_1, \ldots, \mathtt{U}_n$ [Odersky., 2013]. The type constructor T must be a valid type name whereas a type parameter $\mathtt{U}_i$ can either be a specific type value or a type variable. Scala Parameterized Types are similar to Java and C# generics and C++ templates, but express *variance* and *bounds* differently as explained later.

2.5.1.1 Generic Programming

To demonstrate how to use Scala parameterized types to do generic programming, Figure 2.17 gives a simple stack library and an associated client application ported from a Java example found in [Naftalin and Wadler, 2006, Example 5-2]. The example defines an abstract data type Stack, an implementation class `ArrayStack`, a utility method `reverse`, and client application `Client`.

In the example, `Stack` is defined as a *trait*, which is ~~an analogy~~ to an abstract class that supports multiple inheritance. The `Stack` trait defines the signature of three methods: `empty`, `push`, and `pop`. A Stack maintains a collection of data to which an entity can be added (the *push* operation) or be removed (the *pop* operation) in a *Last-In-First-Out* order. The `empty` method defined in the `Stack` trait returns `true` if the collection does not contain any data. The Stack trait takes a type parameter `E` which appears in the push and pop methods as well. The argument of the push method has type `E` so that only data of type `E` can be added to the `Stack`. Consequently, the pop method is expected to return data of type `E`.

```
1  trait Stack[E] {
2    def empty():Boolean
3    def push(elt:E):Unit
4    def pop():E
5  }
6
7  import scala.collection.mutable.ArrayBuffer
8  class ArrayStack[E] extends Stack[E]{
9    private val list:ArrayBuffer[E] = new ArrayBuffer[E]()
10   def empty():Boolean = { return list.size == 0 }
11
12   def push(elt:E):Unit = { list += elt }
13   def pop():E = {
14     val elt:E = list.remove(list.size-1)
15     return elt
16   }
17
18   override def toString():String = {
19     return "stack"+list.toString.drop(11)
20   }
21 }
22
23 object Stacks {
24   def reverse[T](in:Stack[T]):Stack[T] = {
25     val out = new ArrayStack[T]
26     while(!in.empty){
27       val elt = in.pop
28       out.push(elt)
29     }
30     return out
31   }
32 }
33
34 object Client extends App {
35   val stack:Stack[Integer] = new ArrayStack[Integer]
36   var i = 0
37   for(i <- 0 until 4) stack.push(i)
38   assert(stack.toString().equals("stack(0, 1, 2, 3)"))
39   val top = stack.pop
40   assert(top == 3 && stack.toString().equals("stack(0, 1, 2)"))
41   val reverse = Stacks.reverse(stack)
42   assert(stack.empty)
43   assert(reverse.toString().equals("stack(2, 1, 0)"))
44 }
```

Figure 2.17: Scala Example: A Generic Stack Library

The ArrayStack class implements the Stack trait and overrides the toString method which gives a string representation of the Stack. An ArrayStack instance internally saves data in an ArrayBuffer. Prepending an element to an ArrayBuffer (line 8) takes constant time while removing an element from an ArrayBuffer takes time linear regarding to the buffer size.

The utility method reverse repeatedly pops data from one stack and pushes it onto the stack to be returned. Different to Java, Scala classes do not have static members. Therefore the reverse method is defined in a *singleton object*, the only instance of a class with the same name. Notice that, the object Stacks is not type-parameterized, but its method reverse is.

The Client application creates an empty stack of integers, pushes four integers to it, pops out the last one, and then saves the remainder into a new stack in reverse order. The code stack.push(i) takes an advantage of the Scala compiler called *autoboxing*, which converts a primitive type Int to its corresponding object wrapper class Integer. The example code using autoboxing to write cleaner code.

2.5.1.2 Type Bounds

In the above section, we defined a type parameterized stack to which only values whose type is the same as its type variable can be pushed. The benefit is that data popped from the type parameterized stack always has expected type. In a sense, the push(elt:E):Unit method of Stack[E] specified in Figure 2.17 is overly restrictive because it only accepts an argument of type E, but not data of a subtype of E.

Figure 2.18 gives a more flexible Stack, the types of whose elements are either the same as the type parameter or subtypes of the type parameter. In Figure 2.18, the signature of push is changed to push[T<:E](elt:T):Unit, with an additional type parameter T<:E which denotes that T is a subtype of E. In Scala, E is called the *upper bound* of T. Similarly, T>:E means T is a supertype of E and E is called the *lower bound* of T. In Scala, Any is the supertype of all types and Nothing is the subtype of all types.

The remaining code in Figure 2.18 is the same as the code in Figure 2.17 except that, on line 4 of the Client example, the value of i need to be explicitly converted to an Integer because the current Scala compiler is not sophisticated enough.

```scala
trait Stack[E] {
  def empty():Boolean
  def push[T <:  E](elt:T):Unit
  def pop():E
}

import scala.collection.mutable.ArrayBuffer
class ArrayStack[E] extends Stack[E]{
  private val list:ArrayBuffer[E] = new ArrayBuffer[E]()
  def empty():Boolean = {
    return list.size == 0
  }
  def push[T <:  E](elt:T):Unit = {
    list += elt
  }
  def pop():E = {
    val elt:E = list.remove(list.size-1)
    return elt
  }
  override def toString():String = {
    return "stack"+list.toString.drop(11)
  }
}

object Stacks {
  def reverse[T](in:Stack[T]):Stack[T] = {
    val out = new ArrayStack[T]
    while(!in.empty){
      val elt = in.pop
      out.push(elt)
    }
    return out
  }
}

object Client extends App {
  val stack:Stack[Integer] = new ArrayStack[Integer]
  var i = 0
  for(i <- 0 until 4) stack.push(new Integer(i))
  assert(stack.toString().equals("stack(0, 1, 2, 3)"))
  val top = stack.pop
  assert(top == 3 && stack.toString().equals("stack(0, 1, 2)"))
  val reverse = Stacks.reverse(stack)
  assert(stack.empty)
  assert(reverse.toString().equals("stack(2, 1, 0)"))
}
```

Figure 2.18: Scala Example: A Generic Stack Library using Type Bounds

2.5.1.3 Variance Under Inheritance

An important issue that is intentionally skirted in Section 2.5.1.1 is how variance under inheritance works in Scala. Specifically, if T_{sub} is a subtype of T, is `Stack[`T_{sub}`]` the subtype of `Stack[T]`, or reversely? Unlike Java generic collections [Naftalin and Wadler, 2006], which are always invariant on the type parameter, Scala users can explicitly specify one of the three types of variance as part of the type declaration using variance annotation as summarised in Table 2.2, paraphrased from [Wampler and Payne, 2009, Table 12.1].

Variance Annotation	Description
+	Covariant subclassing. i.e. X[T_{sub}] is a subtype of X[T], if T_{sub} is a subtype of T.
-	Contravariant subclassing. i.e. X[T^{sup}] is a subtype of X[T], if T_{sup} is a supertype of T.
default	Invariant subclassing. i.e. cannot substitute X[T^{sup}] or X[T_{sub}] for X[T], if T_{sub} is a subtype of T and T is a subtype of T_{sup}.

Table 2.2: Variance Under Inheritance

A variance annotation constrains positions where the annotated type variable may appear. Specifically, covariant, contravariant, and invariant type variables can only appear in covariant position, contravariant position, and invariant position respectively. The Scala compiler checks if types with variance are used consistently according to a set of rules given in [Odersky., 2013, Section 4.5]. As a programmer, the author of this thesis often find that it is easier to uses variant types according to a variant of the *Get and Put Principle*.

The *Get and Put Principle* for Java Generic Collections [Naftalin and Wadler, 2006, Section 2.4] ~~read as the follows:~~

> **The Get and Put Principle:** *Use an extends wildcard when you only get values out of a structure, use a super wildcard when you only put values into a structure, and don't use a wildcard when you both get and put.*

~~When use generic types with variance in Scala, the general version is:~~

> **The General Get and Put Principle:** *Use a type in covariant positions when you only get values out of a structure, use a type in contravariant positions when you only put values into a structure, and use a type in invariant positions when you both get and put.*

Take the function type for example, a user *puts* an input value into its input channel, *gets* a return value from its output channel. According to the

```scala
trait Stack[+E] {
  def empty():Boolean
  def push[T>:E](elt: T): Stack[T]
  def pop():(E, Stack[E])
}

import scala.collection.immutable.List
class ArrayStack[E](protected val list:List[E]) extends Stack[E]{
  def empty():Boolean = { return list.size == 0 }
  def push[T >:E](elt: T): Stack[T] = { new ArrayStack(elt :: list) }
  def pop():(E, Stack[E]) = {
    if (!empty) (list.head, new ArrayStack(list.tail))
    else throw new NoSuchElementException("pop of empty stack")
  }
  override def toString():String = {
    return "stack"+list.toString.drop(4)
  }
}

object Stacks {
  def reverse[T](in:Stack[T]):Stack[T] = {
    var temp = in
    var out:Stack[T] = new ArrayStack[T](Nil)
    while(!temp.empty){
      val eltStack = temp.pop
      temp = eltStack._2
      out = out.push(eltStack._1)
    }
    return out
  }
}

object Client extends App {
  var stack:Stack[Integer] = new ArrayStack[Integer](Nil)
  var i = 0
  for(i <- 0 until 4) { stack = stack.push(i) }
  assert(stack.toString().equals("stack(3, 2, 1, 0)"))
  stack.pop match {
    case (top, stack) =>
      assert(top == 3 && stack.toString().equals("stack(2, 1, 0)"))
      val reverse:Stack[Integer] = Stacks.reverse(stack)
      assert(reverse.toString().equals("stack(0, 1, 2)"))
      val anystack:Stack[Any] = reverse.push(3.0)
      assert(anystack.toString().equals("stack(3.0, 0, 1, 2)"))
  }
}
```

Figure 2.19: Scala Example: A Covariant Immutable Stack

Is it obvious to you why we have an invariant Stack, a covariant Stack, but not a contraviant Stack?.

General Get and Put Principle, a function is contravariant in the input type and covariant in the output type.

This section concludes with an immutable Stack that is covariant on its type parameter, as shown in Figure 2.19. A stack is covariant on its type parameter because, for example, a stack that saves a collection of `Integer` values is also a stack that saves a collection of `Any` values. However, if the type of Stack is declared as `Stack[+E]`, the signature of its push method *cannot* be

```scala
def push[T<:E](elt:T):Unit
```

while its pop method always returns a value of type `E` ; otherwise, a user can put a value of any type to a stack of integer. The trick is, as shown in the code, making the `Stack[+E]` class an *immutable* collection whose push and pop methods do not modify its content but return a new stack.

2.5.2 Scala Type Descriptors

I'll completely rewrite. I will do or have done?

As in Java, generic types are erased by the Scala compiler. To record type information that is required at runtime but might be erased, Scala users can ask the compiler to keep the type information by using the `Manifest` class.

The Scala standard library contains four manifest classes as shown in Figure 2.20. A `Manifest[T]` encapsulates the runtime type representation of some type `T`. `Manifest[T]` a subtype of `ClassManifest[T]`, which declares methods for subtype (<:<) test and supertest (>:>). The object `NoManifest` represents type information that is required by a paraterized type but is not available in scope. `OptManifest[+T]` is the supertype of `ClassManifest[T]` and `OptManifest`.

```scala
package scala.reflect
trait OptManifest[+T] extends Serializable

object NoManifest extends OptManifest[Nothing] with Serializable

trait ClassManifest[T] extends OptManifest[T] with Serializable
  def <:<(that: ClassManifest[_]): Boolean
  def >:>(that: ClassManifest[_]): Boolean
  erasure: Class[_]

trait Manifest[T] extends ClassManifest[T] with Serializable
```

Figure 2.20: Scala API: Manifest Type Hierarchy

```
1  package sample.other
2
3  import scala.reflect._
4
5  object ManifestExample extends App {
6    assert(List(1,2.0,"3").isInstanceOf[List[String]])
7    // Compiler Warning :non-variable type argument String in type
8        List[String] is unchecked since it is eliminated by
9    // erasure
10
11   case class Foo[A](a: A)
12   type F = Foo[_]
13   assert(classManifest[F].toString.equals(
14           "sample.other.ManifestExample$Foo[<?>]"))
15   assert(NoManifest.toString.equals("<?>"))
16
17   assert(manifest[List[Int]].toString.equals(
18           "scala.collection.immutable.List[Int]"))
19   assert(manifest[List[Int]].erasure.toString.equals(
20           "class scala.collection.immutable.List"))
21
22   def typeName[T](x: T)(implicit m: Manifest[T]): String = {
23     m.toString
24   }
25   assert(typeName(2).equals("Int"))
26
27   def boundTypeName[T:Manifest](x: T):String = {
28     manifest[T].toString
29   }
30   assert(boundTypeName(2).equals("Int"))
31
32   def isSubType[T: Manifest, U: Manifest] = manifest[T] <:< manifest[U]
33   assert(isSubType[List[String], List[AnyRef]])
34   assert(! isSubType[List[String], List[Int]])
35 }
```

Figure 2.21: Scala ExampleI: Manifest Example

The code example in Figure 2.21 shows common usages of Manifest. There are three ways of obtaining a manifest: using the Methods `manifest` (line 16) or `classManifest` (line 12), using an implicit parameter of type `Manifest[T]` (line 21), or using a `context` bound of a type parameter (line 26). Context bound can be seen as a syntactic sugar for implicit parameters without a user-specified parameter name. The `isSubType` method defined at line 31 tests if the first `Manifest` represents a type that is a subtype of the typed represented by the second `Manifest`.

2.6 Summing Up

To review, the Actor Model [Hewitt et al., 1973] is proposed for designing concurrent systems. It is employed by Erlang [Armstrong, 2007b] and other programming languages. Erlang developers designed the Supervision Principle in 1999 when the Erlang/OTP library is released as an open-source project. With the supervision principle, actors are supervised by their supervisors, who are responsible for initializing and monitoring their children. Erlang developers claimed that applications that using the supervision principle have achieved a high availability [Armstrong, 2002]. Recently, the actor programming model and the supervision principle have been ported to Akka, an Actor library written in Scala. Although Scala is a statically typed language and provides a sophistical type system, the type of messages sent to Akka actors are dynamically checked when they are processed. The next chapter presents the design and implementation of the TAkka library where type checks are involved in the earliest opportunity to expose type errors.