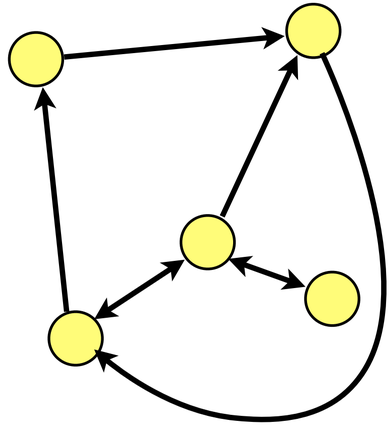
Message Passing Systems

- Actors and message passing
- Structure of communication
- Implementation in Scala and Rust

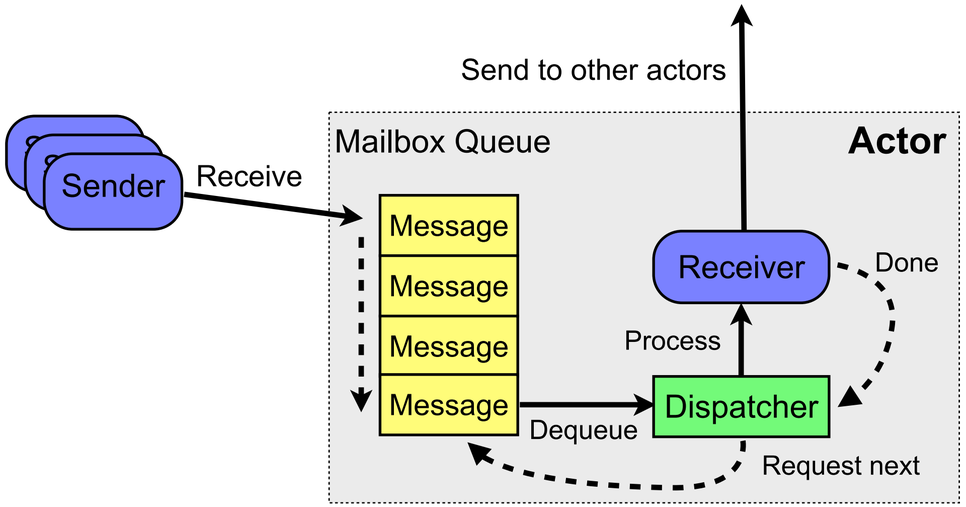
Message Passing Systems

- System is structured as a set of communicating processes, *actors*, with no shared mutable state
- All communication via exchange of messages
  - Messages are generally required to be immutable – data conceptually copied between processes
  - Some systems use linear types to ensure messages are not referenced after they are sent, allowing mutable data to be safely transferred
- Implementation
  - Implementation within a single system usually built with shared memory and locks, passing a reference to the message – rely on correct locking of message passing implementation
  - Trivial to distribute, by sending the message down a network channel – the runtime needs to know about the network, but the application can be unaware that the system is distributed



Message Handling

- Receivers pattern match against messages
  - Match against message types, not just values
  - Type system can ensure an exhaustive match
- Messages queued for processing
  - Dispatcher manages a thread pool servicing receiver components of the actors
  - Receivers operate in message processing loop – single-threaded, with no concern for concurrency
  - Sent messages enqueued for processing by other actors



Types of Message Passing

- Several different message passing system designs:
  - Synchronous vs asynchronous
  - Statically or dynamically typed
  - Direct or indirect message delivery

- Each has advantages and disadvantages

Types of Message Passing: Interaction Models

- Message passing can involve rendezvous between sender and receiver
  - A synchronous message passing model – sender waits for receiver
- Alternatively, communication may be asynchronous
  - The sender continues immediately after sending a message
  - Message is buffered, for later delivery to the receiver
  - Synchronous rendezvous can be simulated by waiting for a reply

Types of Message Passing: Typed Communication

- Statically-typed communication
  - Explicitly define the types of message that can be transferred
  - Compiler checks that receiver can handle all messages it can receive – robustness, since a receiver is guaranteed to understand all messages
- Dynamically-typed communication
  - Communication medium conveys any time of message; receiver uses pattern matching on the received message types to determine if it can respond to the messages
  - Potentially leads to run-time errors if a receiver gets a message that it doesn't understand

Types of Message Passing: Naming

- Are messages sent between named processes or indirectly via channels?
  - Some systems directly send *messages* to actors (processes), each of which has its own mailbox
  - Others use explicit *channels*, with messages being sent indirectly to a mailbox via a channel

  - Explicit channels require more plumbing, but the extra level of indirection between sender and receiver may be useful for evolving systems
  - Explicit channels are a natural place to define a communications protocol for statically typed messages

Implementations

- Message passing starting to see wide deployment, with two widely used architectures:
  - Dynamically typed with direct delivery
    - Erlang programming language (https://www.erlang.org/)
    - Scala programming language (http://www.scala-lang.org) and Akka library (http://akka.io)
    - Dynamically typed – any type of message may be sent to any receiver
    - Messages sent directly to named actors, not via channels
    - Both provide transparent distribution of processes in a networked system
  - Statically typed, with explicit channels
    - Rust programming language (https://www.rust-lang.org/)
    - Use asynchronous statically typed messages passed via explicit channels

Example: Scala+Akka

```
import akka.actor.Actor
import akka.actor.ActorSystem
import akka.actor.Props

class HelloActor extends Actor {
  def receive = {
    case "hello" => println("hello back at you")
    case _       => println("huh?")
  }
}

object Main extends App {
  // Initialise actor runtime
  val runtime = ActorSystem("HelloSystem")

  // Create an actor, running concurrently
  val helloActor = runtime.actorOf(Props[HelloActor], name = "helloactor")

  // Send it some messages
  helloActor ! "hello"
  helloActor ! "buenos dias"
}
```

The actor comprises a receive loop that reacts to messages as they're received

Complete program is a collection of actors that exchange messages

Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```

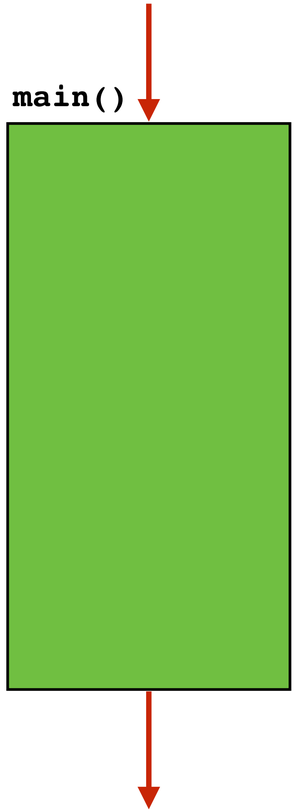
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



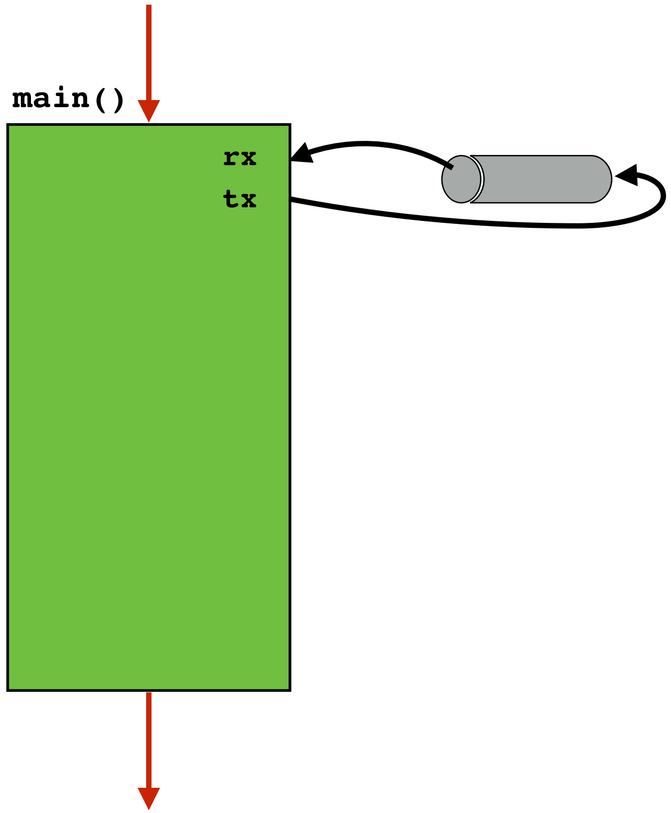
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



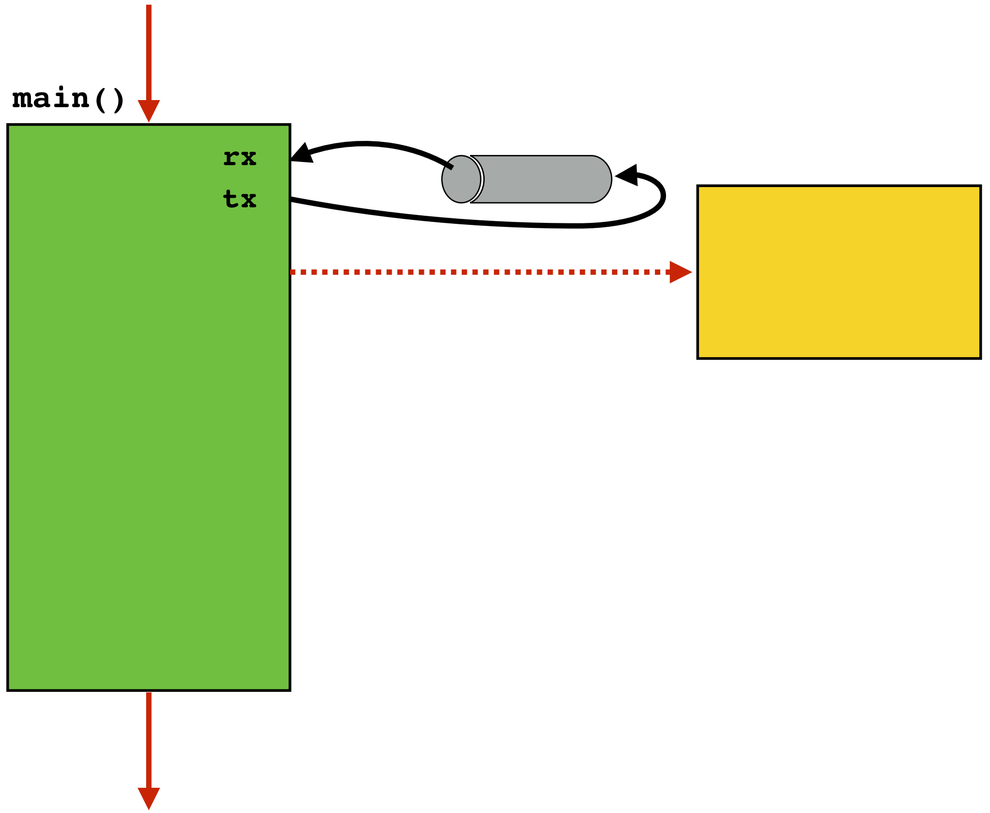
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



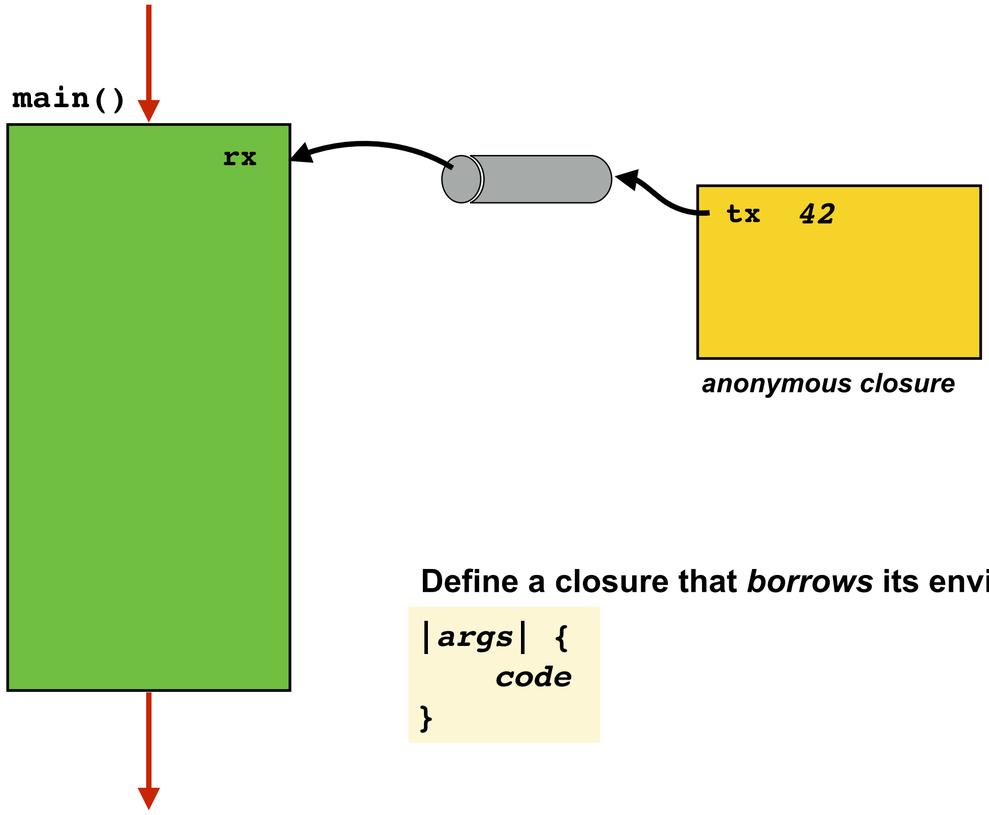
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



Define a closure that *borrows* its environment:

```
|args| {
    code
}
```

Define a closure that *takes ownership of* its environment:

```
move |args| {
    code
}
```

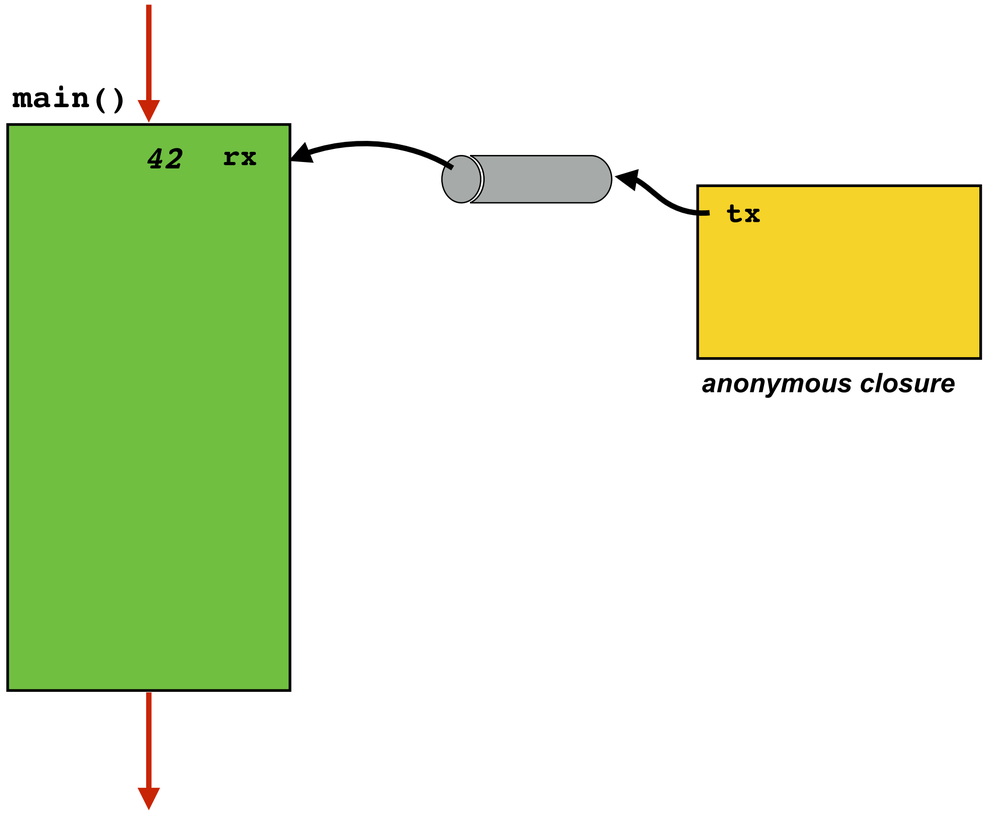
Example: Rust

```
use std::sync::mpsc::channel;
use std::thread;

fn main() {
  let (tx, rx) = channel();

  thread::spawn(move|| {
    let _ = tx.send(42);
  });

  match rx.recv() {
    Ok(value)  => {
      println!("Got {}", value);
    }
    Err(error) => {
      // An error occurred…
    }
  }
}
```



Trade-offs

- The two approaches behave quite differently:
  - **Scala+Akka** let weakly coupled processes to communicate via asynchronous and dynamically typed messages:
    - Expressive, flexible, and extensible actor model
    - Robust framework for error handling via separate processes
    - Relative ease of upgrading running systems via dynamic actor insertion
    - Checking happens at run time, so guarantees of robustness are probabilistic
  - **Rust** uses statically typed message passing provides compile-time checking that a process can respond to messages
    - But, requires more plumbing to connect channels
    - Has more explicit error handling
  - The usual static vs. dynamic typing debate

Message Passing Systems

- Actors and message passing
- Immutable data
- Ownership and race conditions