Lecture 12

Previously, in cs2030..

Final Exam

- **Open Book:** bring your notes!
- APIs (not in the notes) will be provided
- **Covers Lecture 1 - 12, Lab 0 - 6** with more emphasis on topics not covered in midterm

- MCQs with short questions
- Mixed of conceptual and programming questions

Abstractions

Abstractions

- Abstraction of data: types
- Abstraction of instructions: methods
- Composite data type: classes and objects

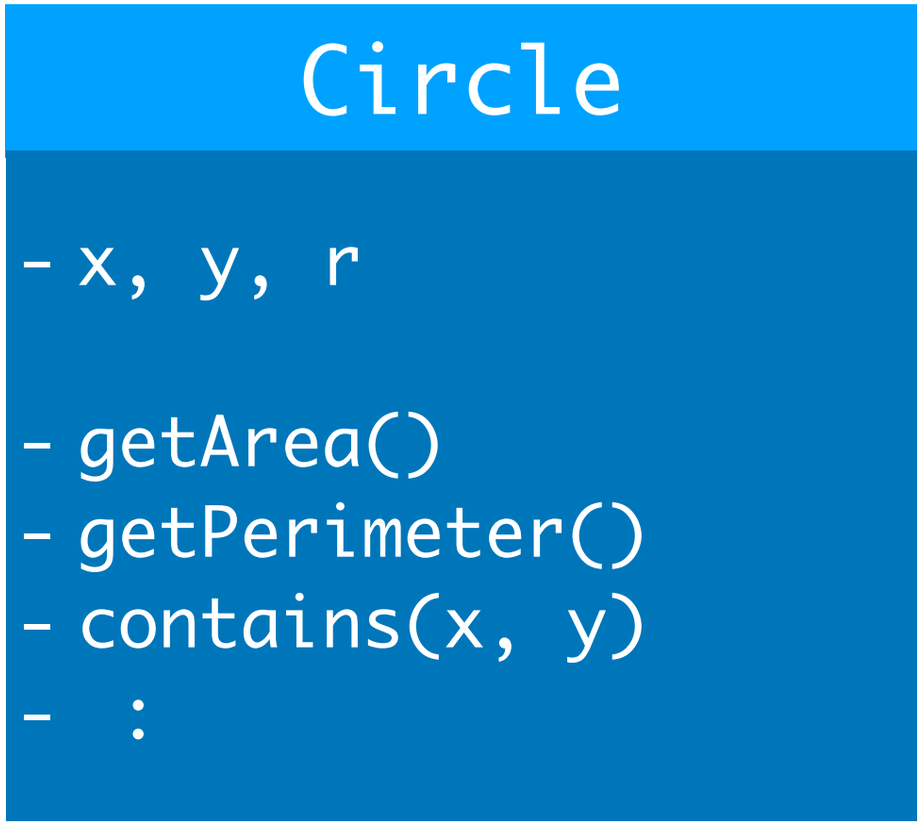
Abstractions

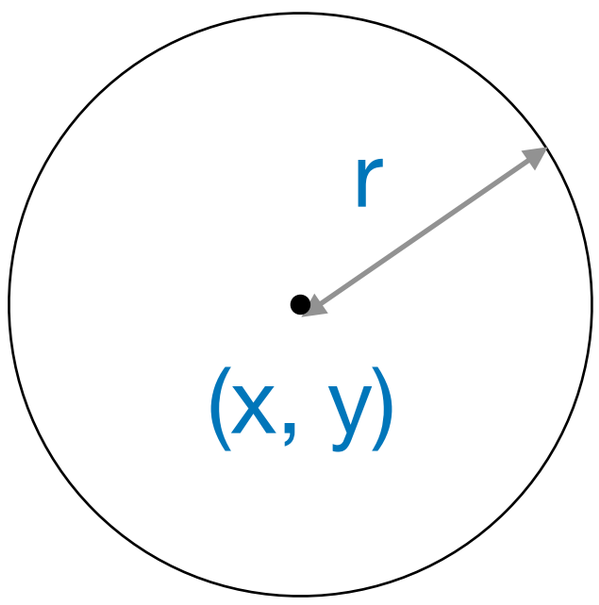
- Abstraction of common behaviours across different types: generic types
- Abstraction of common behaviours over different behavior: lambdas

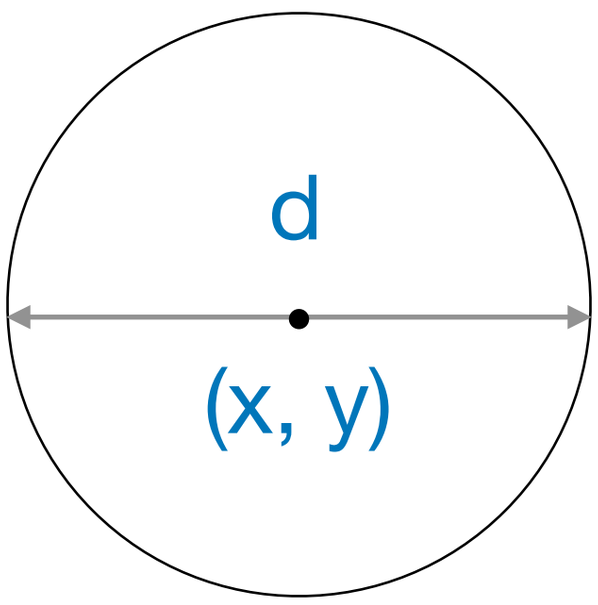
Abstractions

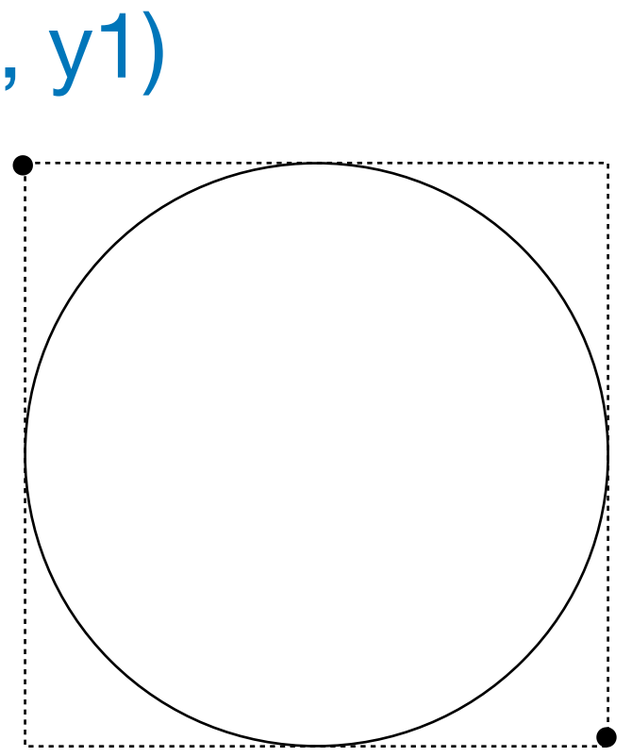
- One of the most important concepts in computer science for dealing with complexity (in our case, complex code)
- You will see abstractions in computer architecture, OS, DB, networking, etc.

Encapsulation: Grouping data and related methods together in a class.


Circle
- x, y, r
- getArea()
- getPerimeter()
- contains(x, y)
- :








```
area = 3.1416*r*r;
```

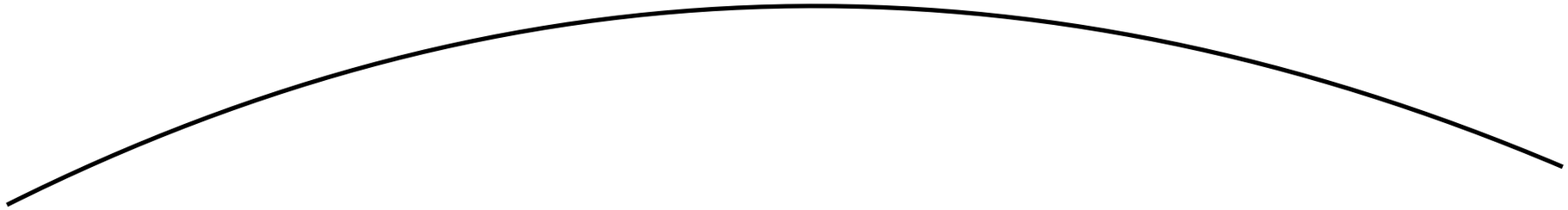
```
area = 0.7854*d*d;
```

```
area = …
```

Abstraction Barrier

usage of circle

client



implementation of circle

implementer

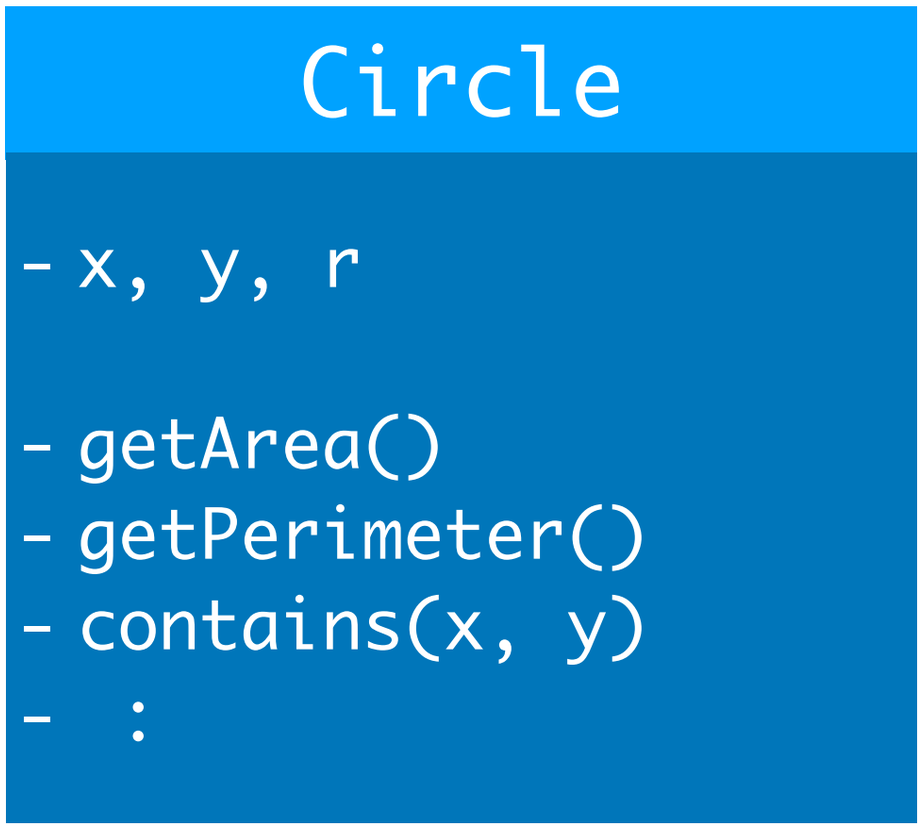
New mindset:

Write code for others

Data Hiding

private ->

public ->



Avoid getters and setters

"If we need to know the internal and do something with it, we are doing it wrong. The right approach is to implement a method within the class that do whatever we want the class to do." - Lecture 1

```
// getter for radius
double circumference = 2*c.getR()*3.1415926;
```

```
// better
double circumference = c.getCircumference();
```

In Lab 1b: Many did this in either Event or Customer class.

```
ArrayList<Server> servers = shop.getServers();
Server idleServer = null;
for (Server s : servers) {
   Customer c = s.getCurrentCustomer();
   if (c == null) {
      idleServer = s;
      break;
   }
}
```

You need to know that

- getCurrentCustomer returns a Customer reference or a null (not an id or -1)
- getServers() returns an ArrayList, not a LinkedList

This is better

```
List<Server> servers = shop.getServers();
Server idleServer = null;
for (Server s : servers) {
  if (s.isIdle()) {
     idleServer = s;
     break;
   }
}
```

You still need to know that

- getServers() returns a List, not an array

Note: this code does not make use of any fields
from Event or Customer or Simulator

```
List<Server> servers = shop.getServers();
Server idleServer = null;
for (Server s : servers) {
  if (s.isIdle()) {
     idleServer = s;
     break;
   }
}
```

This is better

```
Server server = shop.getFirstIdleServer();
```

Type

- Java is a strongly typed language
- Java is a statically typed language
- Subtyping
- Type conversion
- Variance of type
- Generic type

Use the type system wisely to catch bugs at **compile** time, not runtime.

```
String state = server.getState();
switch (state) {
   case “Idle”:
     doSomething();
  case “Busy”:
     doSomethingElse();
```

Type Safe

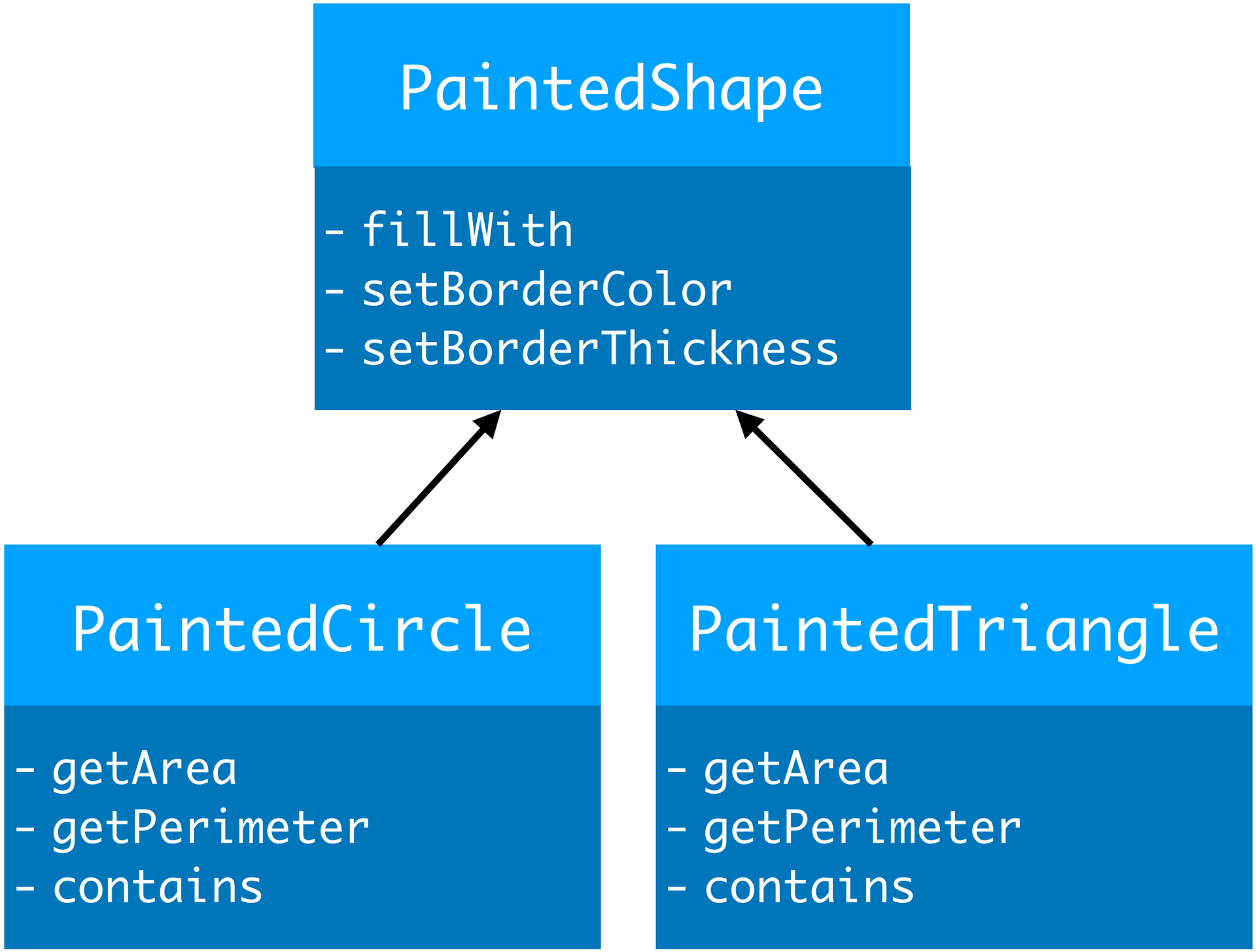
```
ServerState state = server.getState();
switch (state) {
  case ServerState.IDLE:
    doSomething();
case ServerState.BUSY:
    doSomethingElse();
```

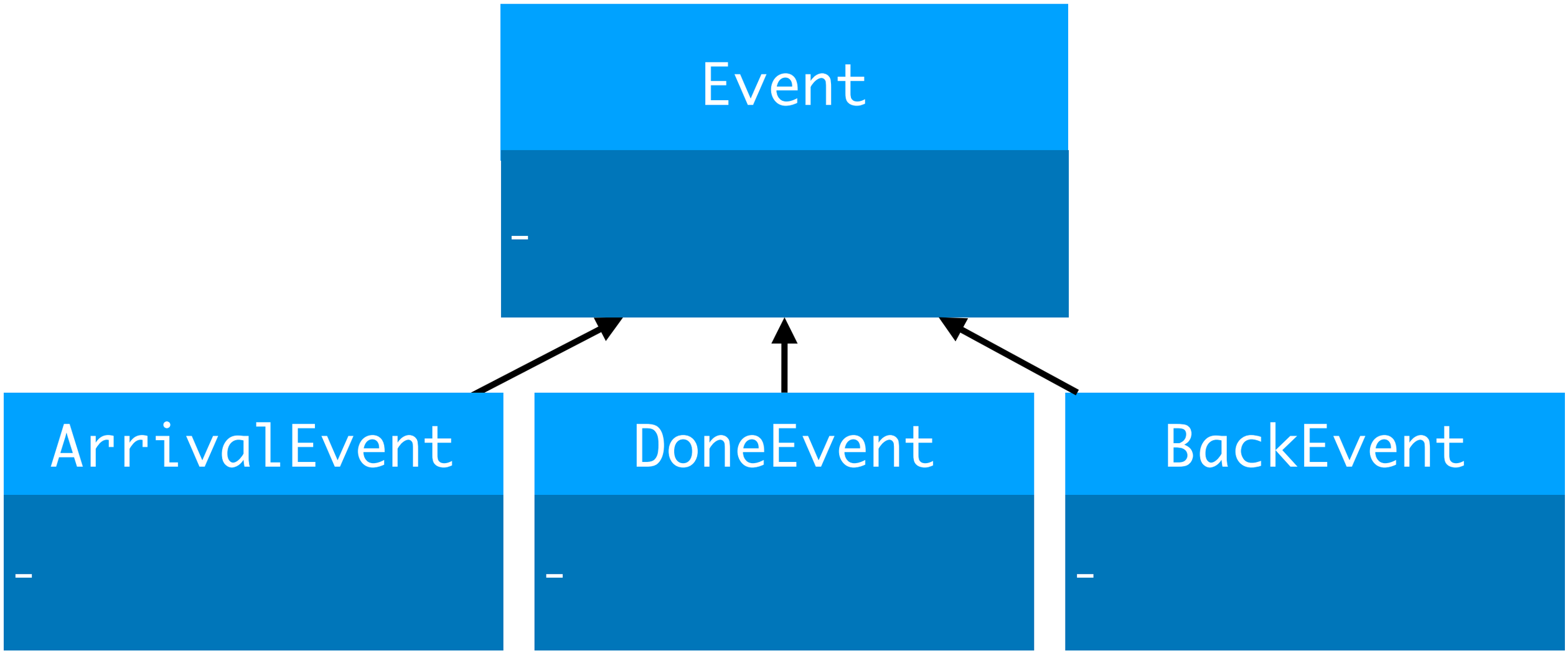
```
DoneEvent(double time,
      int serverId,
      int customerId) {
      :
      :

}
```

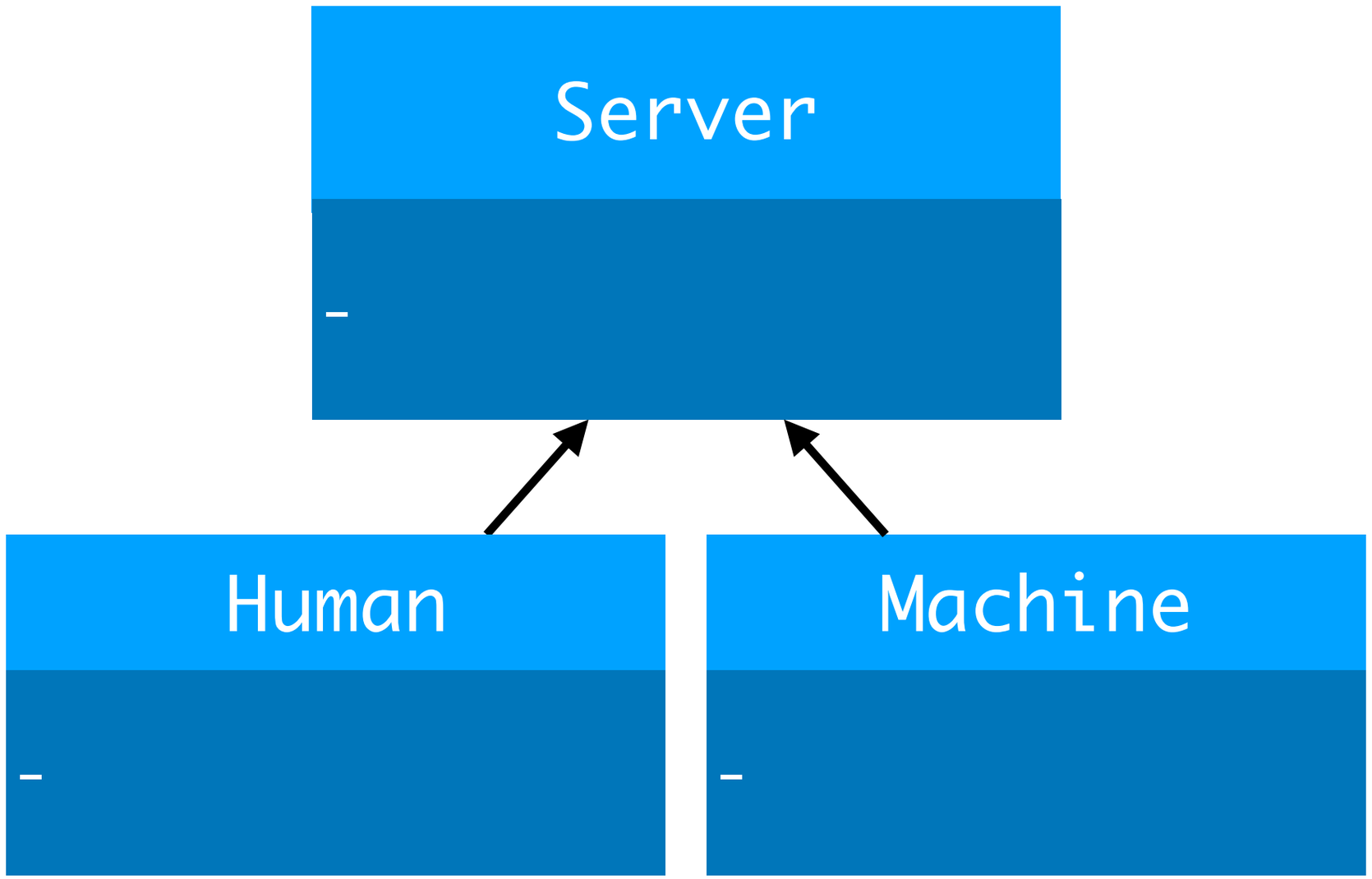
Type Safe

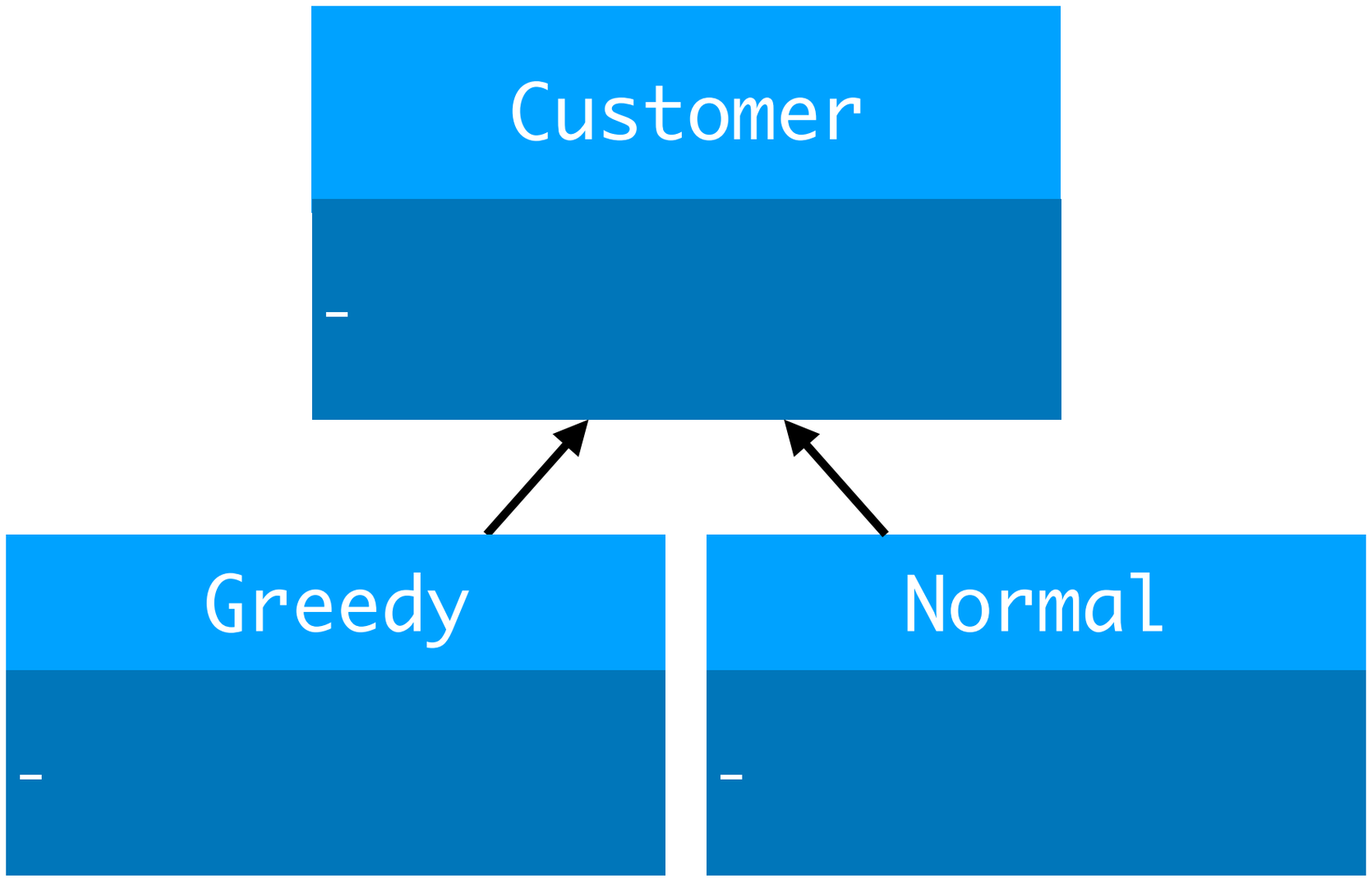
```
DoneEvent(double time,
    Server server,
    Customer customer) {
    :
    :
}
```

Inheritance: Abstract out common properties/methods into a common parent class.


PaintedShape
- fillWith
- setBorderColor
- setBorderThickness
PaintedCircle
- getArea
- getPerimeter
- contains
PaintedTriangle
- getArea
- getPerimeter
- contains


Event
-
ArrivalEvent
-
DoneEvent
-
BackEvent
-

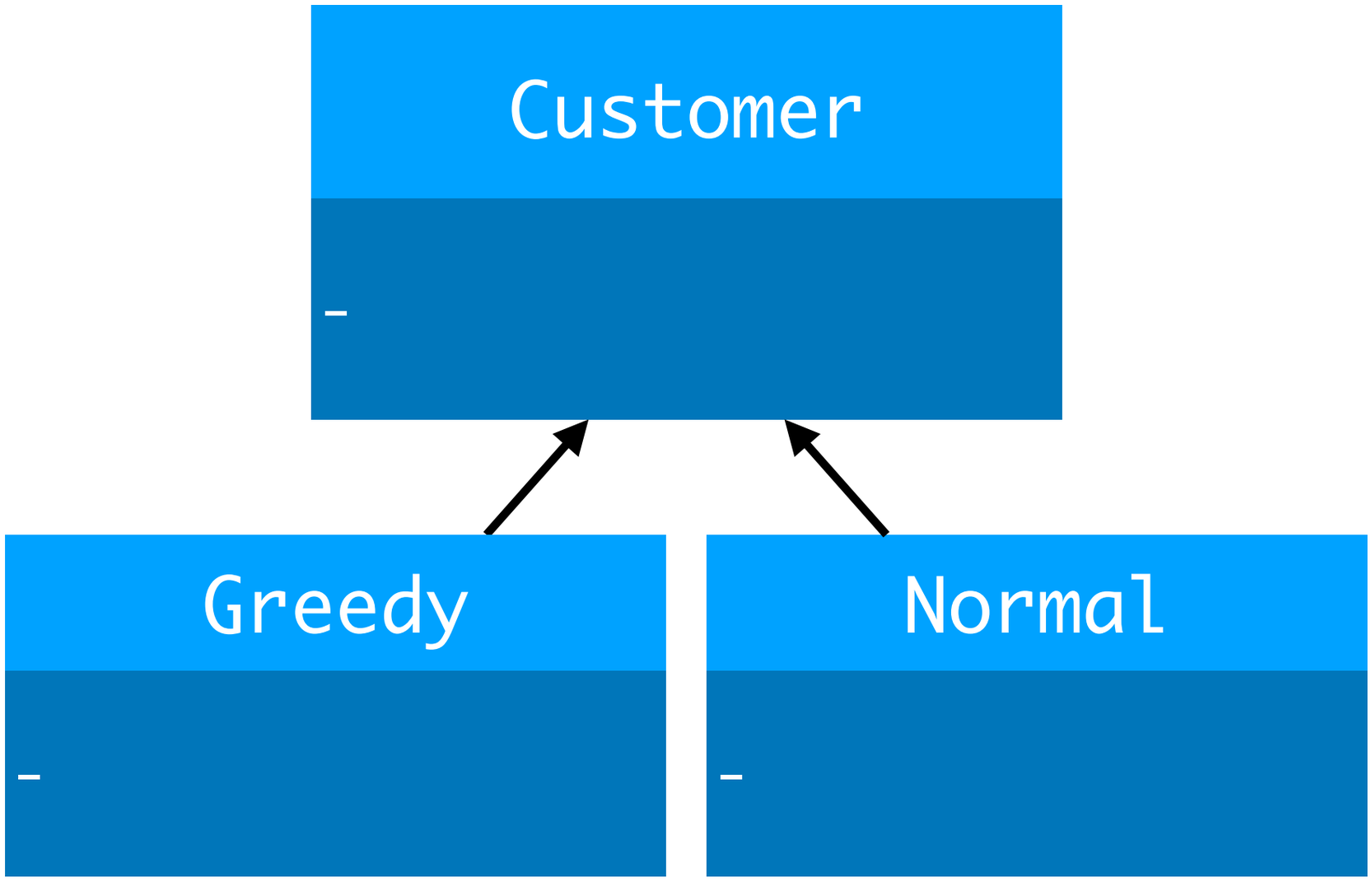

Server
-
Human
-
Machine
-


Customer
-
Greedy
-
Normal
-

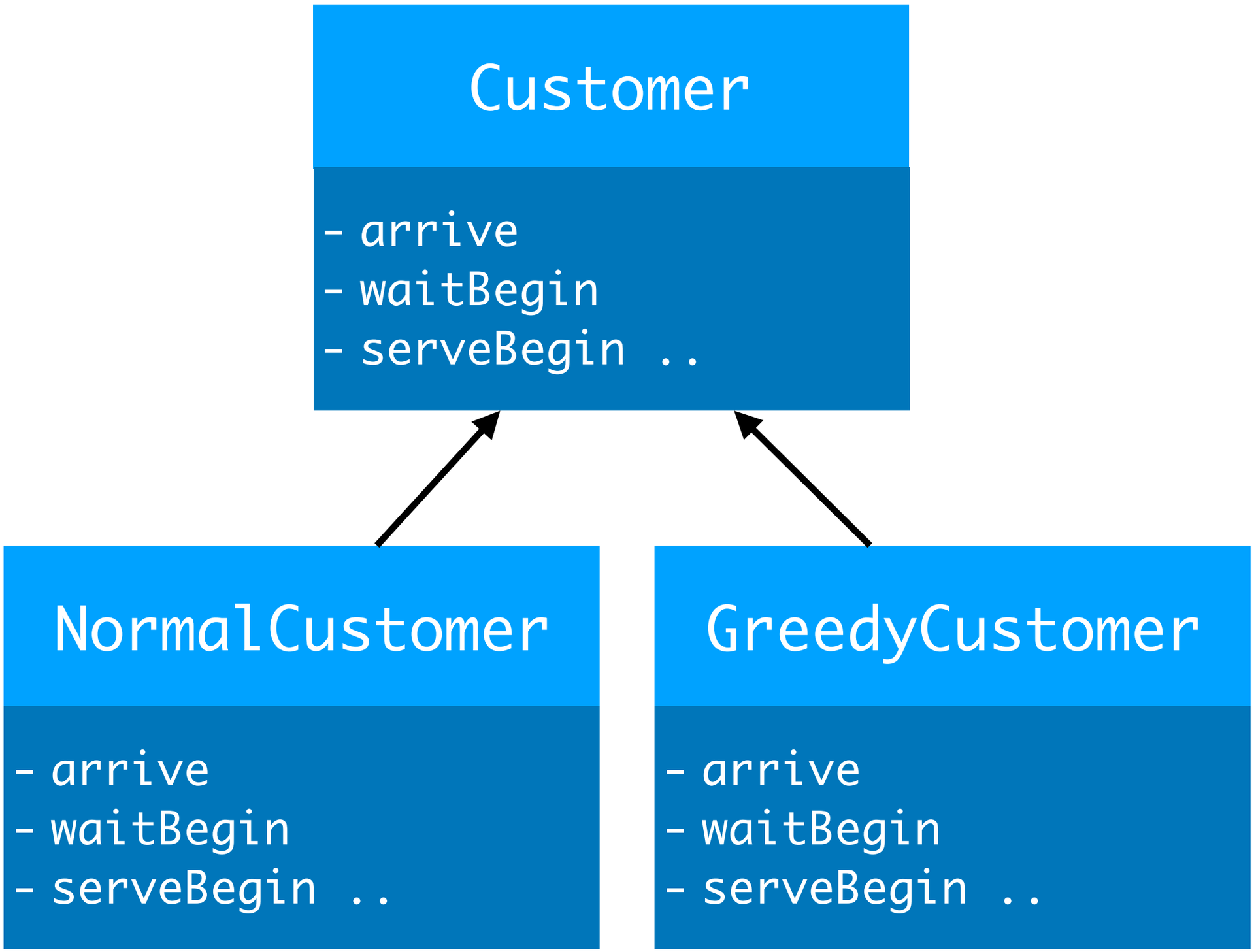
"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."

The Abstraction Principle

Varying parts ->



Unfortunately, in Lab 2b



(Subtype) **Polymorphism:**

Behavior of an object depends on its type during run-time

```
for (Printable o: objs) {
    o.print();
}
```

```
for (Printable o: objs) {
  if (o instanceof Circle) {
    o.printCircle();
  } else if (o instanceof Point) {
    o.printPoint();
  }
}
```

What if we need to add more printables?

```
for (Printable o: objs) {
  if (o instanceof Circle) {
    o.printCircle();
  } else if (o instanceof Point) {
    o.printPoint();
  } else if (o instanceof Square) {
    :
  } else if ( ..
}
```

```
for (Printable o: objs) {
    o.print();
}
```

Unfortunately, in Lab 2b

```
if (s instanceof HumanServer) {
   s.shouldRest();
       :
}
```

```
if (c instanceof GreedyCustomer) {
   c.findShortestQueue()
else if (c instanceof NormalCustomer) {
   c.findFirstAvailableQueue();
}
```

(Subtype) **Polymorphism:**

Behavior of an object depends on its type during run-time

Other Types of Polymorphism

- ad hoc polymorphism (aka method overloading)
- parametric polymorphism (aka generics in Java)

Other Types of Polymorphism

- ad hoc polymorphism (aka method overloading)
- parametric polymorphism (aka generics in Java)

S.O.L.I.D. Principles

Liskov Substitution Principle

(SOID in CS2103)

OO Design Patterns

The Observer Pattern

The Strategy/Policy Pattern

(The rest in CS2103/CS3219)

Composition: HAS-A

Inheritance: IS-A

New mindset:

Write code for others (incl your future self)

- Provide a clear interface
- Hide your implementation details
- Provide documentation

- Make your code clean and readable, following a certain convention
- Allow extension by inheritance

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live”

- John F. Wood

Modern Programming Constructs

- Optional to avoid null
- Future/Promises to represent data that is not yet available
- Monads to chain and manipulate variables in certain context
- Threads/tasks as abstraction for concurrent programming

Effect-free Programming

- Pure vs. unpure functions
- Immutable data types
- No side effects
- Made parallelization simple

Lazy evaluation

- Asynchronous programming
- Infinite data structures

Anonymous Class
SAM
Lambda Expression

```
new Comparator<Event>() {
   public int compare(Event e1, Event e2) {
      return e1.earlierThan(e2);
   }
}
```

```
Function<Integer,Integer> sqr = new
    Function<Integer,Integer>() {
      public Integer apply(Integer x) {
        return x * x;
    }
};
```

```
Function<Integer,Integer> incr = new
    Function<Integer,Integer>() {
      public Integer apply(Integer x) {
        return x + 1;
      }
};
```

```
Function<Integer,Integer> sqr = new
    Function<Integer,Integer>() {
      public Integer apply(Integer x) {
        return x * x;
      }
};
```

```
Function<Integer,Integer> sqr =
    (Integer x) -> {
      return x * x;
};
```

```
Function<Integer,Integer> sqr =
    (Integer x) -> {
      return x * x;
};
```

```
Function<Integer,Integer> sqr =
    x -> x * x;
```

```
Function<Integer,Integer> sqr =
    x -> x * x;
Function<Integer,Integer> incr =
    x -> x + 1;
```

```
Function<Integer,Integer> sqr =
    x -> x * x;
Function<Integer,Integer> incr =
    x -> x + 1;
sqr.apply(4);
incr.apply(5);
```

- Predicate<T>
  - boolean test(T t)

- Supplier<T>
  - T get()

- Consumer<T>
  - void accept(T t)

- BiFunction<T,U,R>
  - R apply(T t, U u)

Lambdas can be

- partially evaluated
- evaluated later
- passed around
- composed during run time

Infinite stream of values

- filter
- map
- reduce
- ⋮

```
void fiveHundredPrime() {
    int count = 0;
    int i = 2;
    while (count < 500) {
        if (isPrime(i)) {
            System.out.println(i);
            count++;
        }
        i++;
    }
}
```

```
void fiveHundredPrimes() {
   IntStream.iterate(2, x -> x+1)
            .filter(x -> isPrime(x))
            .limit(500)
            .forEach(System.out::println);
}
```

```
Shop(int numOfServers) {
   this.servers = Stream
      .iterate(0, i -> i + 1)
      .map(i -> new Server(i))
      .limit(numOfServers)
      .collect(Collectors.toList());
}
```

getWaitingCustomer of a server

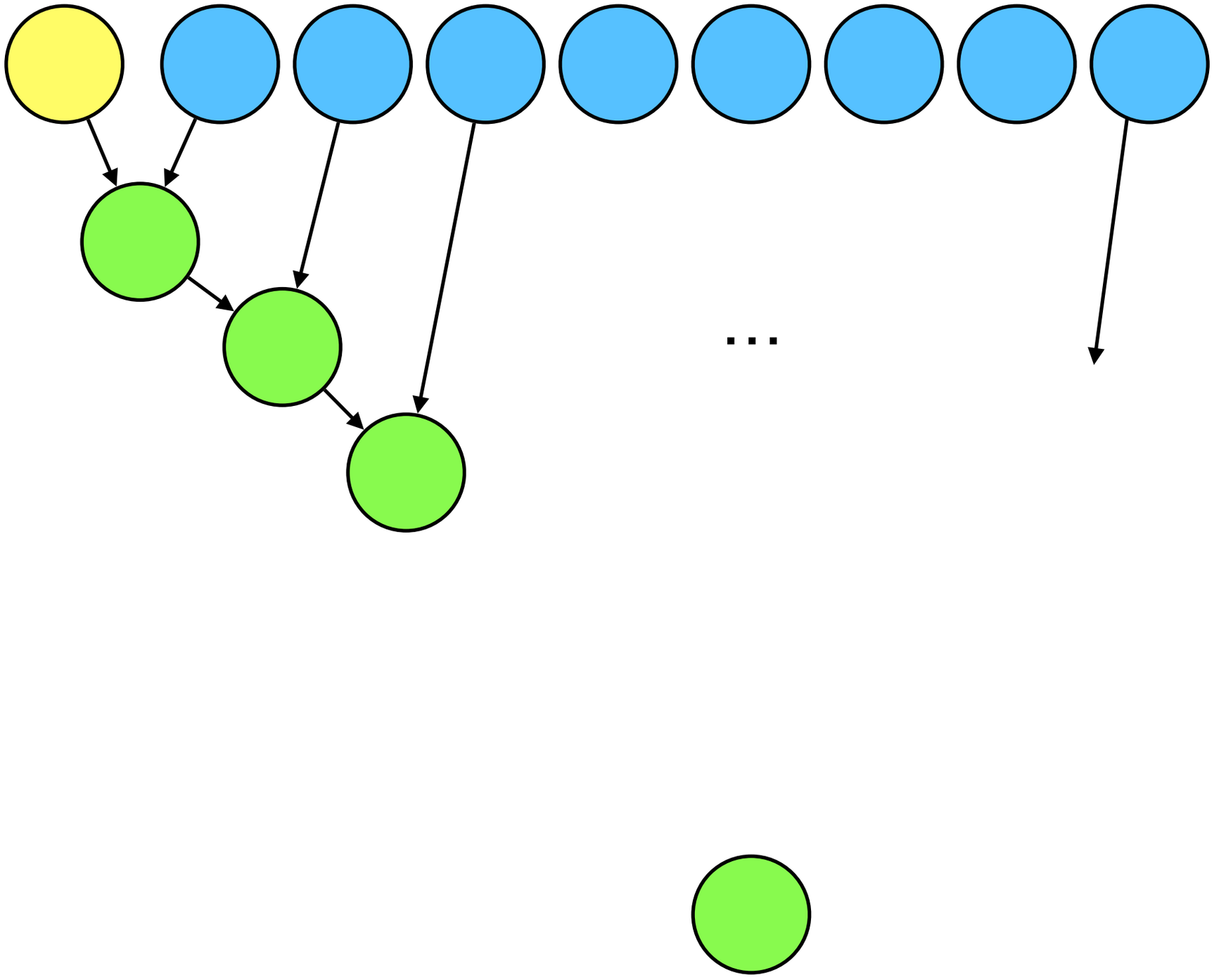
```
return servers.stream()
        .filter(s -> s.equals(server))
        .findFirst()
        .flatMap(s -> s.getWaitingCustomer());
```

update a server

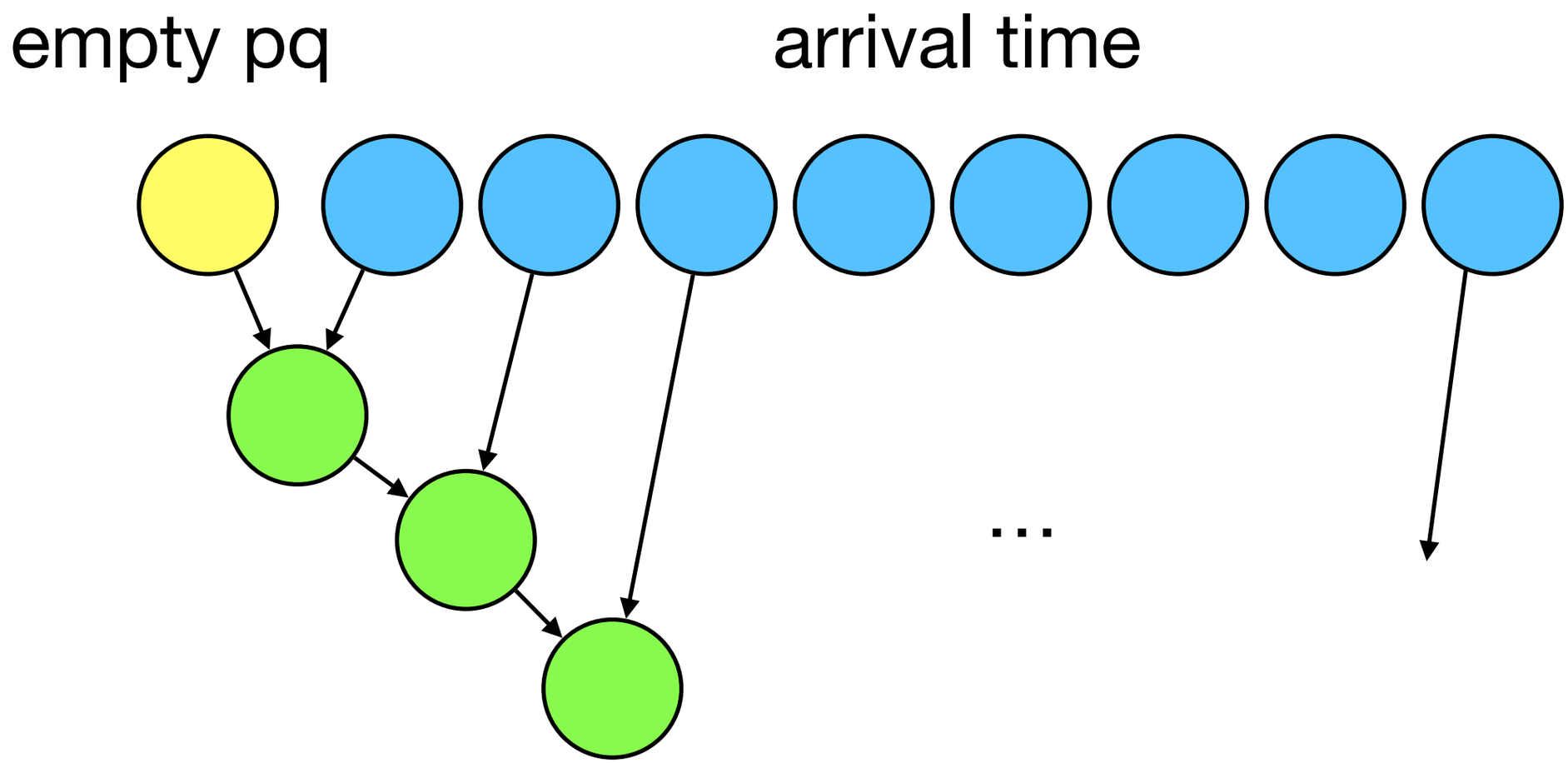
```
return new Shop(servers.stream()
        .map(s -> (s.equals(server) ? server : s))
        .collect(Collectors.toList()));
```

```
scanner.tokens()
    .map(line -> Double.parseDouble(line))
    .reduce(sim.state,
        (state, time) -> state.addEvent(time,
            s -> s.simulateArrival(time)),
        (x, y) -> x)
.run();
```

reduce

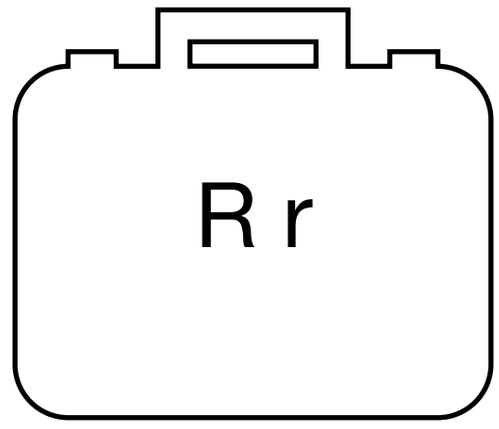


reduce



Monad & Functor

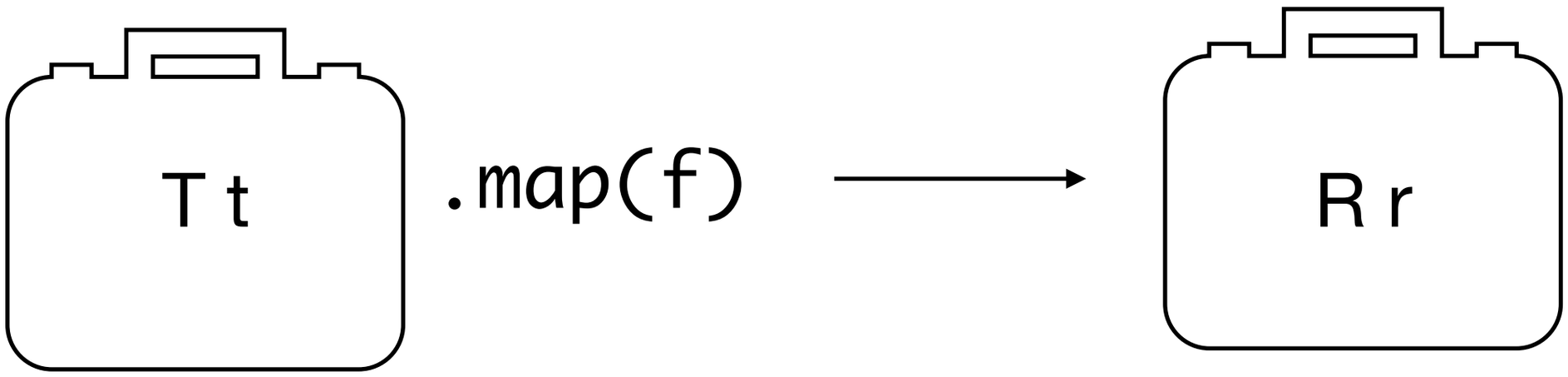
Explained in OO way



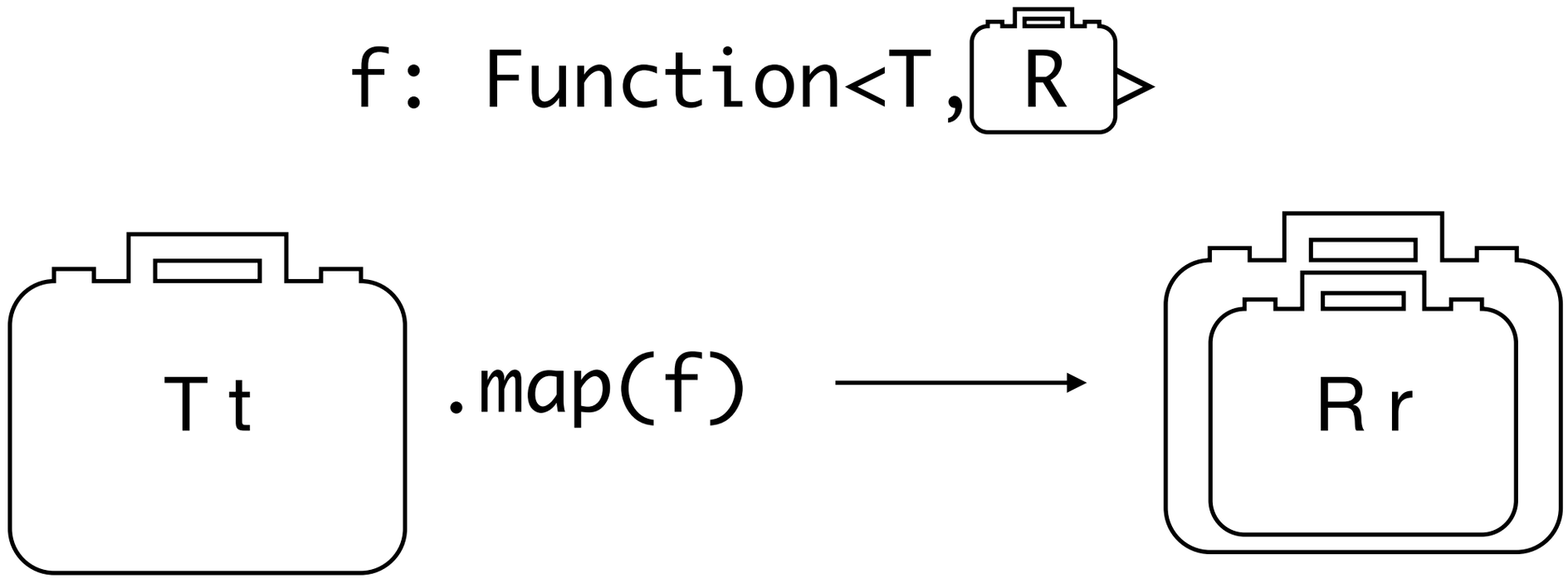
Thing(s) in a box (briefcase) in a certain context

“modify things in a box”

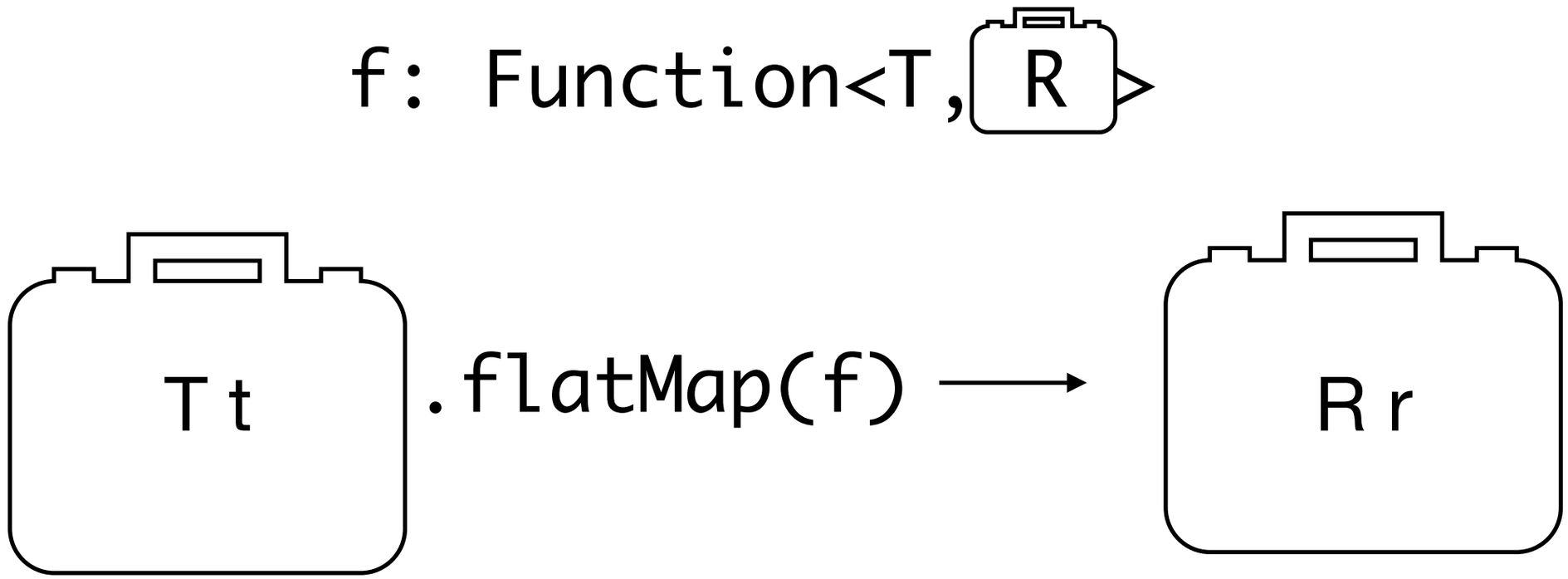
```
f: Function<T, R>
```



Functor



Monad



Monad

Explained in OO way

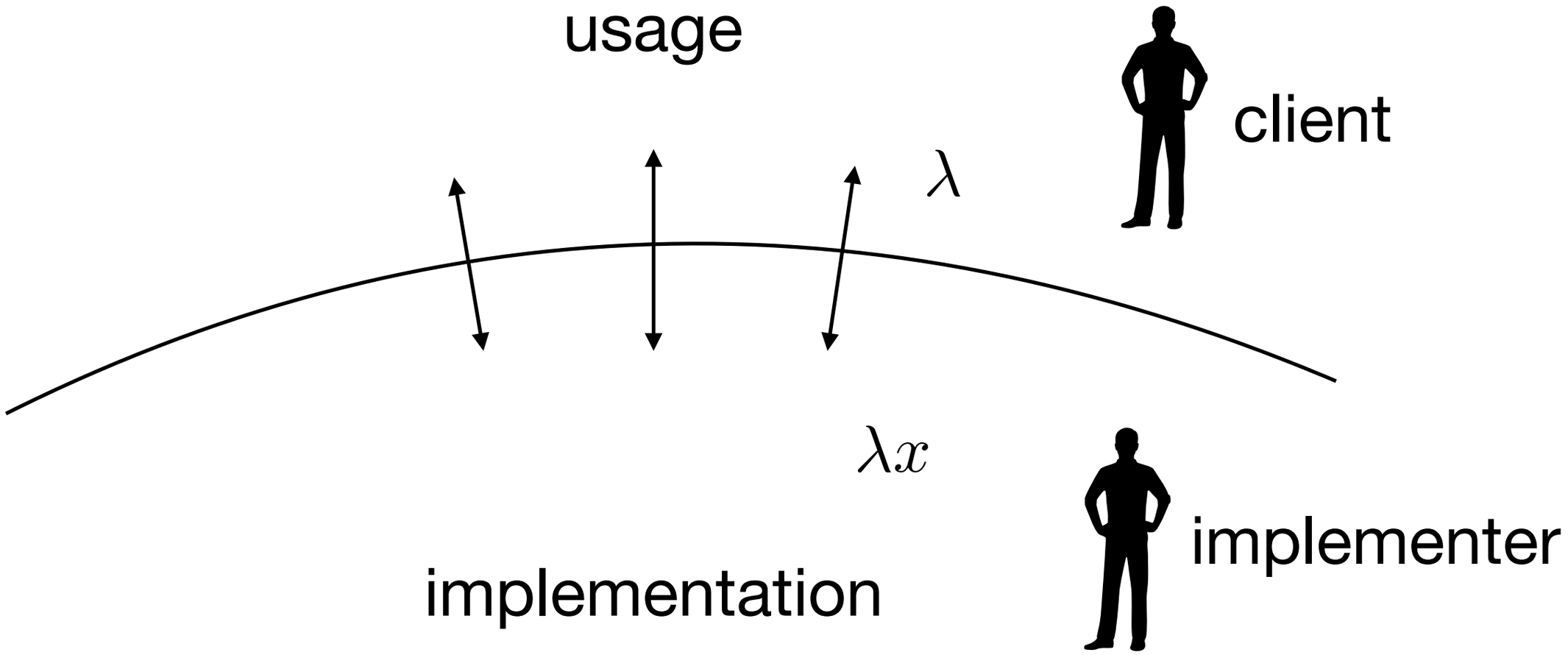
Class	Context / Box
Stream	A sequence of values
Optional	A value that maybe there
CompletableFuture	A value that will be available in the future

Let’s build a Monad

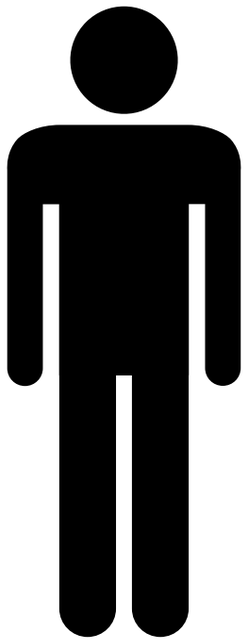
Monad

Explained in OO way

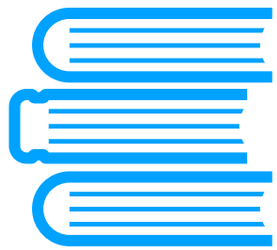
Class	Context / Box
Logger	A value with logged history
Stream	A sequence of values
Optional	A value that maybe there
CompletableFuture	A value that will be available in the future


usage
client
λ
λx
implementation
implementer

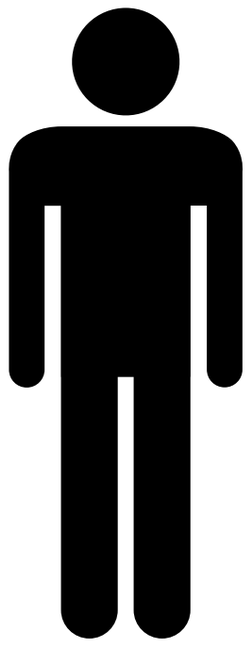
Fork and Join

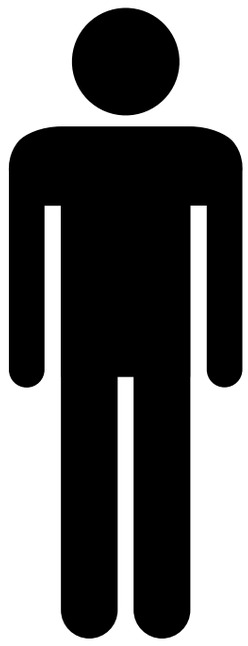


boss



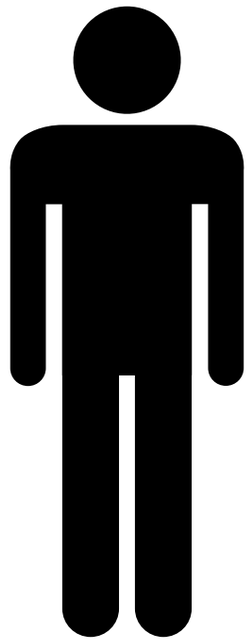
left-hand man



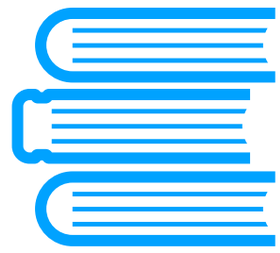


right-hand man

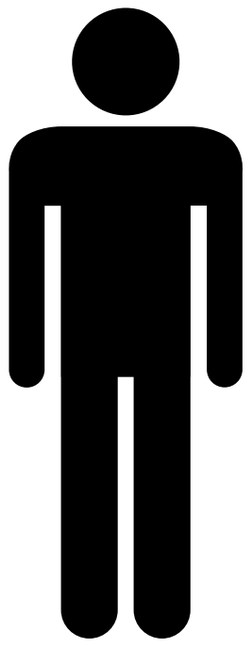
do my homework for me!

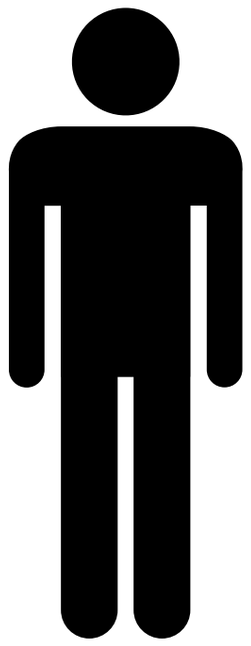


boss

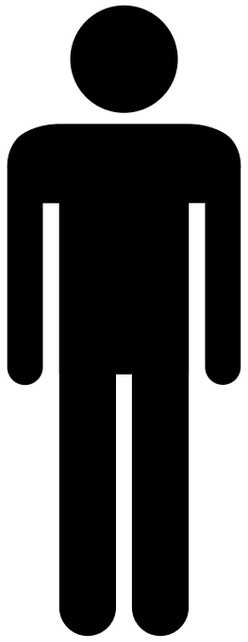


left-hand man





right-hand man



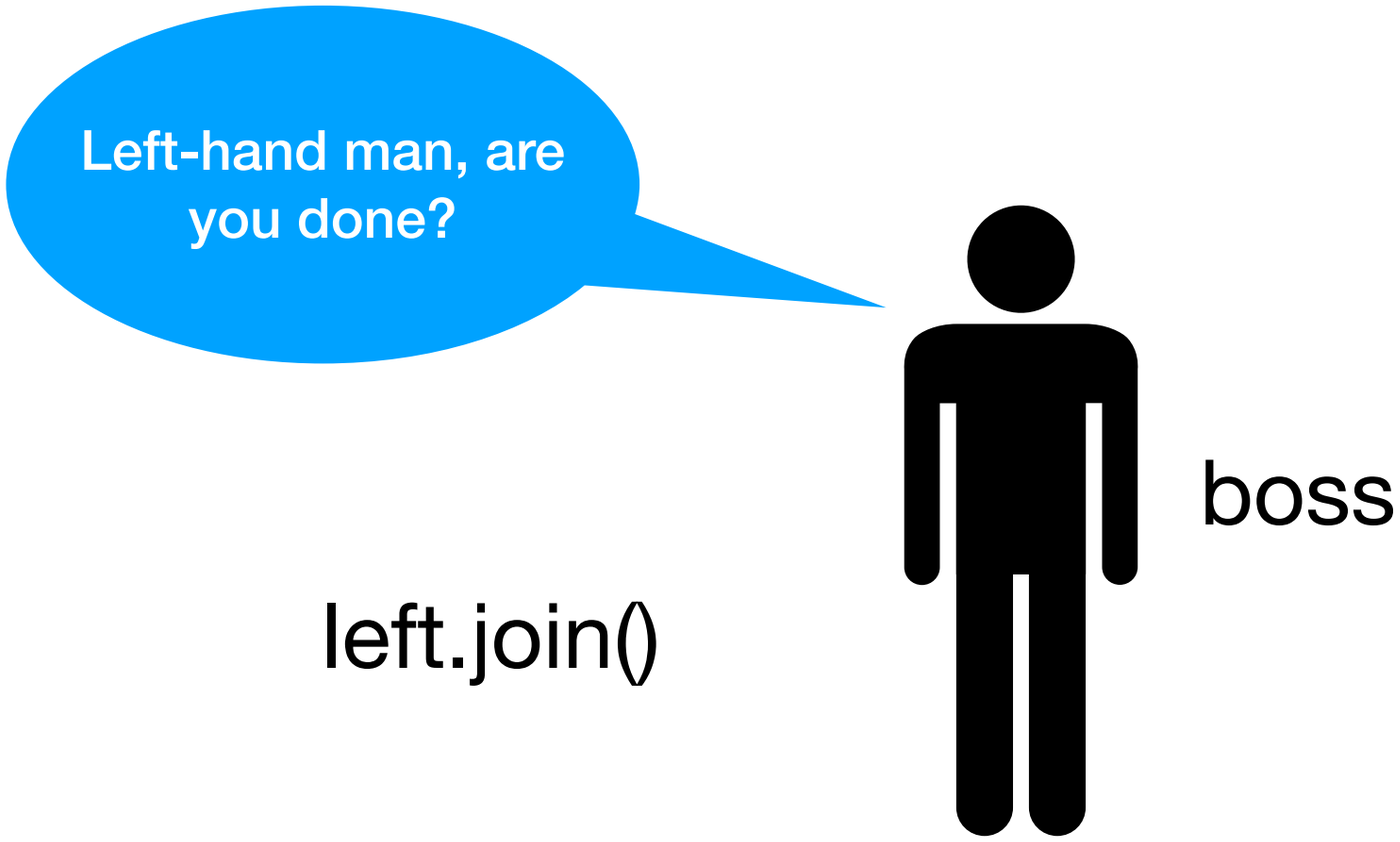
boss

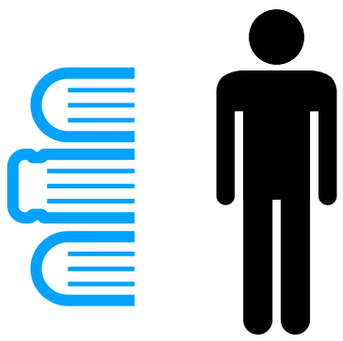
right away!

ok, boss!

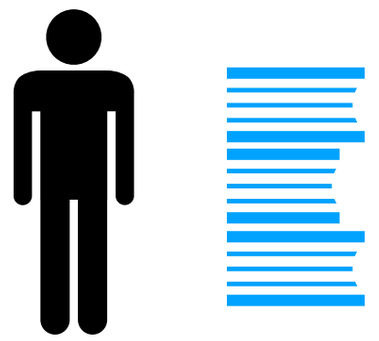
left.fork()

right.fork()

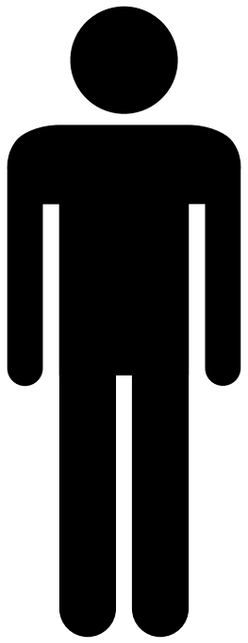




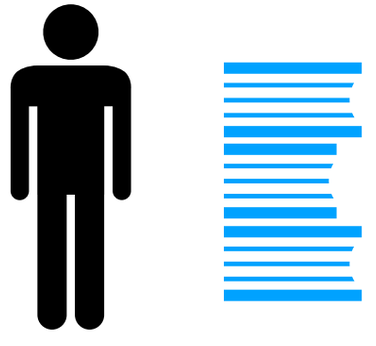
(working concurrently)

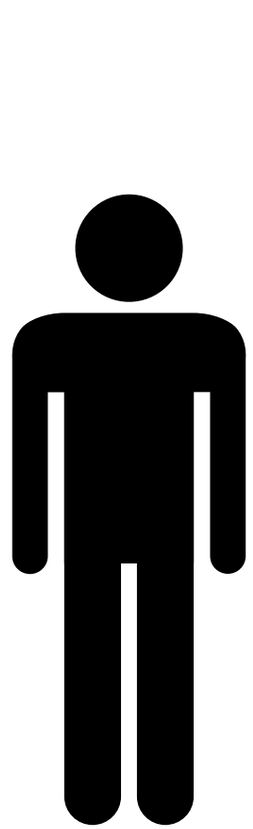


left.join()

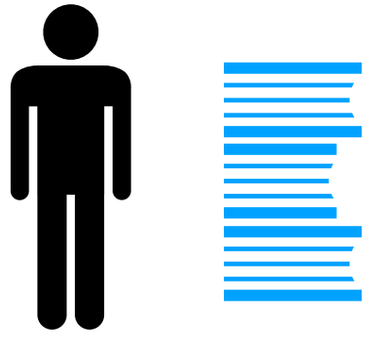


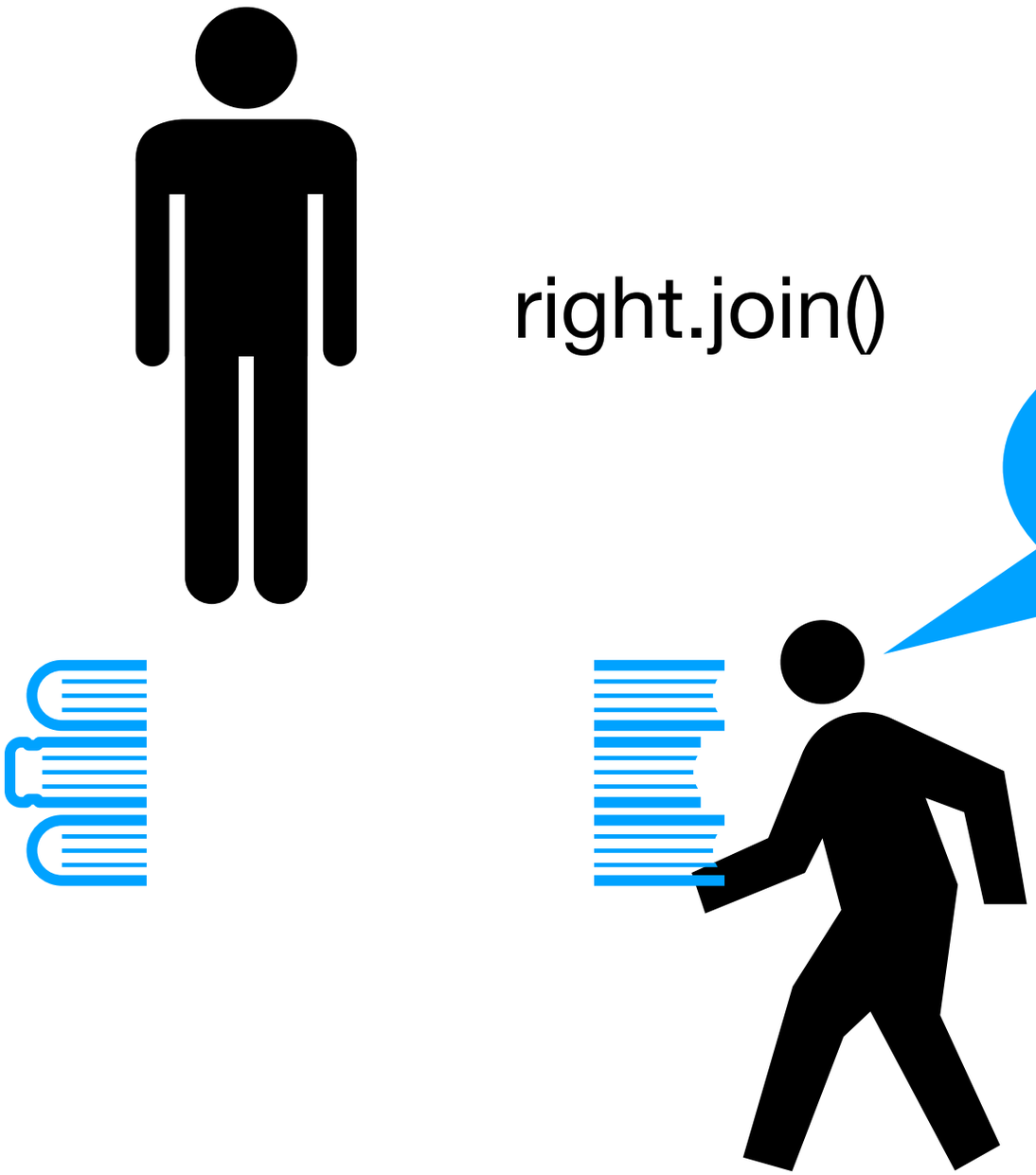
boss



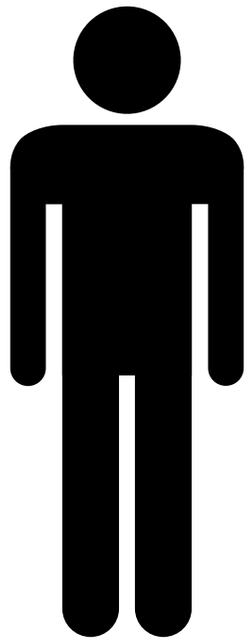


right.join()

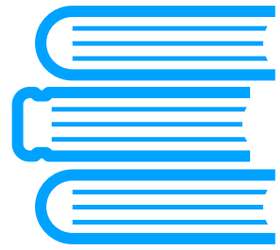



right.join()
Here it is.

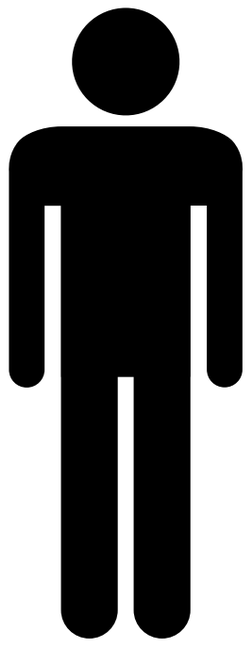
do my homework for me!

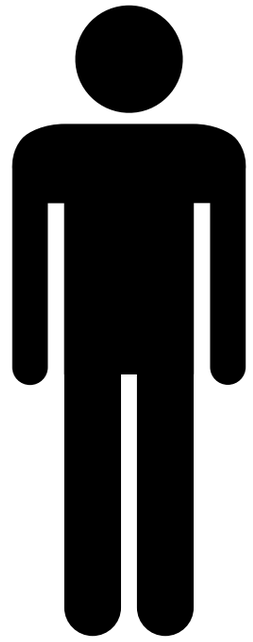


boss



left-hand man

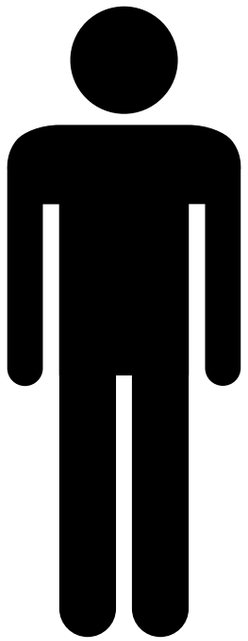


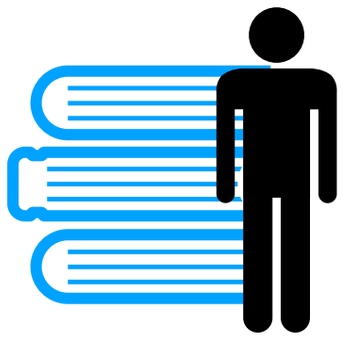


right.compute()

right away!

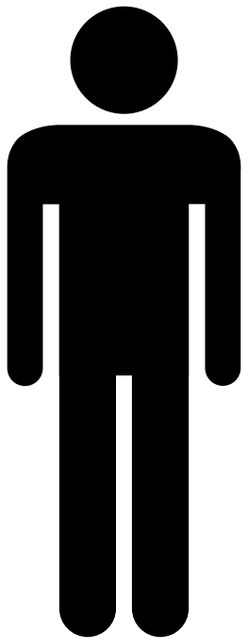
left.fork()



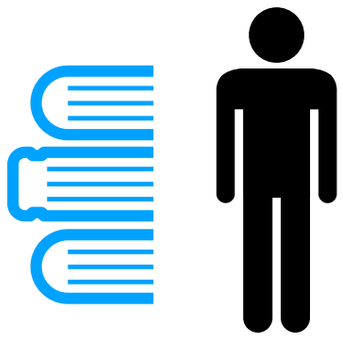


(working concurrently)

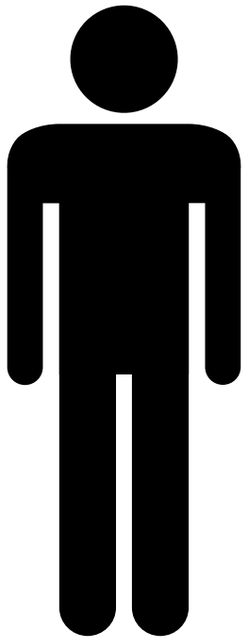
I am done. How about you, left?



left.join()



left.join()



boss

OOP vs FP

- We are not telling you
  - “OOP is the best way to write your code!”
  - nor “FP is the best way to write your code!”

- We are just saying, “here are the different, possibly better ways, to write your code.”

- OOP's strength is polymorphism (abstractions over function pointers)
- FP's strength is immutability

Michael Feathers
@mfeathers

Follow

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

8:27 AM - 3 Nov 2010

434 Retweets **371** Likes

6 434 371

“OO programming is good, when you know what it is. Functional programming is good when you know what it is. And functional OO programming is also good once you know what it is.”

- Uncle Bob , OO vs FP

Java vs Others

- You don’t always have a choice in programming language
- But if you do, there are many possibilities

- **Functional**: Haskell, Erlang, OCaml
- **Concurrent / Parallel**: Erlang, Go
- **Multi-paradigm**: Javascript, Kotlin, Clojure, Scala, Goovy etc.

- Have to use Java? Consider
  - Google Guava
  - Apache Commons
  - Functional Java / JOOλ

What’s Next after CS2030 ?

CS2103

Intro to Software Enginering

CS2104

Concepts of Programming Languages

CS3210/11

Parallel Computing
Parallel / Concurrent Programming

Conclusion

- I hope CS2030 has:
  - level-up your software development skills and experience
  - expand your mind about different ways of writing programs

imperative
programming

object
oriented

functional
programming