Purely Functional State

Goals

- Purely functional random number generation
- Working with stateful APIs
- The State data type

To write purely functional programs that manipulate state, using the simple domain of **random number generation***.*

Learn the basic pattern for how to make **any** *stateful API* purely functional.

Random Number Generation w/ Side Effects

```
scala> val rng = new scala.util.Random

scala> rng.nextDouble
res1: Double = 0.9867076608154569

scala> rng.nextDouble
res2: Double = 0.8455696498024141

scala> rng.nextInt
res3: Int = -623297295

scala> rng.nextInt(10)
 res4: Int = 4
```

Creates a new random number generator seeded with the current system time

Gets a random integer between 0 and 9

APIs are stateful …
so they are not referentally transparent

- Not testable
- Not composable
- Not modular,
- Not easily parallelized

Tests need to be reproducible

```
def rollDie: Int =
  val rng = new scala.util.Random
  rng.nextInt(6) // off by one error
```

- Cure?

```
def rollDie(rng: scala.util.Random): Int = rng.nextInt(6)
```

The “same” generator has to be both created with the same seed, and also be in the same state, which is difficult to guarantee.

Purely functional Generation

- The key to recovering referential transparency is to make the state updates explicit.
- Don’t update the state as a side effect.

```
trait RNG:
  def nextInt: (Int, RNG)
```

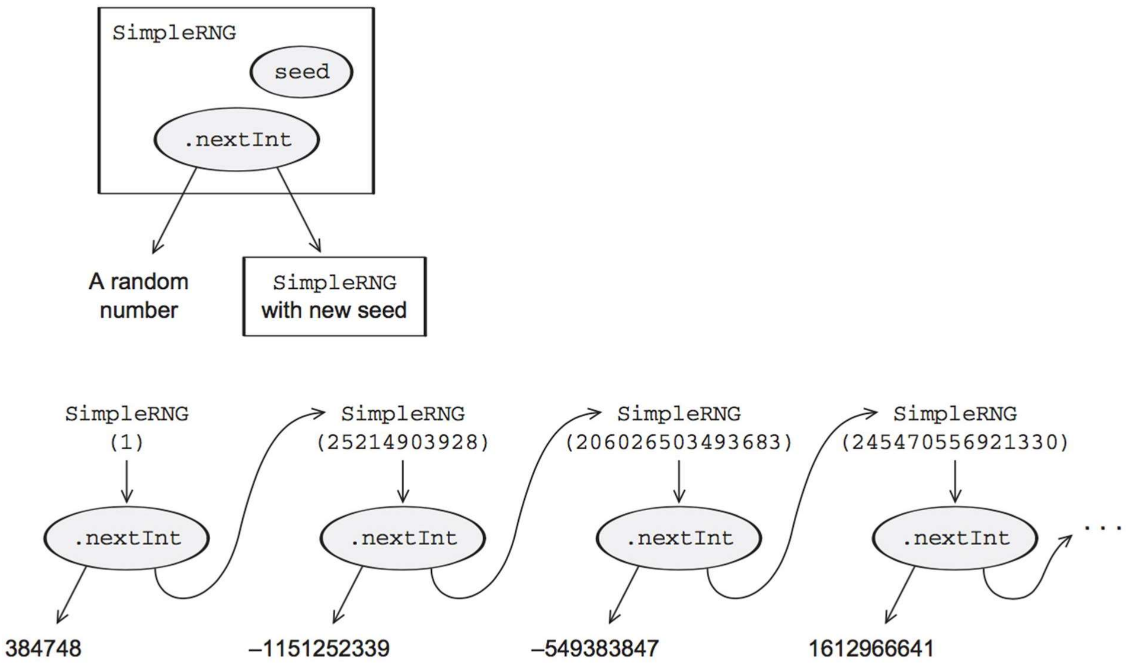
- Separate the concern of computing what the next state is from the concern of communicating the new state to the rest of the program.

A purely functional random number generator

```scala
/*
 * linear congruential generator
 */
case class SimpleRNG(seed: Long) extends RNG:
  def nextInt: (Int, RNG) =
    val newSeed = (seed * 0x5DEECE66DL + 0xBL) & 0xFFFFFFFFFFFFL
    val nextRNG = SimpleRNG(newSeed)
    val n = (newSeed >>> 16).toInt
    (n, nextRNG)
```

The next state, which is an RNG instance created from the new seed.

The return value is a tuple containing both a pseudo-random integer and the next RNG state.



Each call to `SimpleRNG.nextInt` returns the next random number in the sequence and the `SimpleRNG` object needed to continue the sequence.

Using Pure API

```
scala> val rng = SimpleRNG(42)
rng: SimpleRNG = SimpleRNG(42)

scala> val (n1, rng2) = rng.nextInt
n1: Int = 16159453
rng2: RNG = SimpleRNG(1059025964525)

scala> val (n2, rng3) = rng2.nextInt
n2: Int = -1281479697
rng3: RNG = SimpleRNG(197491923327988)
```

Let's choose an arbitrary seed value, 42.

Syntax for declaring two values by deconstructing the pair returned by `rng.nextInt`.

Common Pattern

- To make pure API, make explicit the transition from one state to the next.

```
class Foo:
  private var s: FooState = ...
  def bar: Bar // somehow modify s
  def baz: Int // somehow modify s, too
```

```
trait Foo:
  def bar(FooState): (Bar, FooState)
  def baz(FooState): (Int, FooState)
```

- Now, the caller is responsible for passing the computed next state through the rest of the program.

```
def randomPair(rng: RNG): ((Int, Int), RNG) =
  val (i1, _) = rng.nextInt
  val (i2, rng2) = rng.nextInt
  ((i1, i2), rng2)
```

What's the difference?

```
def randomPair(rng: RNG): ((Int, Int), RNG) =
  val (i1, rng2) = rng.nextInt
  val (i2, rng3) = rng2.nextInt
  ((i1, i2), rng3)
```


Try to find common patterns in usages

- [Exercise 6.1]

```
def nonNegativeInt(rng: RNG): (Int, RNG) = ???
```

- [Exercise 6.2]

```
def double(rng: RNG): (Double, RNG) = ???
```

- [Exercise 6.3]

```
def intDouble(rng: RNG): ((Int, Double), RNG) = ???
def doubleInt(rng: RNG): ((Double, Int), RNG) = ???
d4f double3(rng: RNG): ((Double, Double, Double), RNG) = ???
```

- [Exercise 6.4]

```
def ints(count: Int)(rng: RNG): (List[Int], RNG) = ???
```

Better API for State Action

- Each of our functions has a type of the form **RNG => (A, RNG)** for some type **A**.
- Functions of this type are called **state actions** or **state transitions** because they transform RNG states from one to the next.
- These state actions can be combined using **combinators** through which to pass the state from one action to the next *automatically*.

Type Alias for Convenience

```
type Rand[+A] = RNG => (A, RNG)
```

- Informally, a value of type **Rand[A]** as “a randomly generated A.”
- It’s really a state action—a *program* that depends on some RNG, uses it to generate an **A**, and also transitions the RNG to a new state that can be used by another action later.

```
def int(rng: RNG): (Int, RNG) = rng.nextInt
```

```
val int: Rand[Int] = rng => rng.nextInt
```

We want to write functions that let us combine Rand actions while avoiding explicitly passing along the RNG state.

We'll end up with a kind of domain specific language (DSL) that does all of the passing for us.

pure (or unit or return)

A simple RNG state transition which passes the RNG state through without using it, always returning a constant value rather than a random value:

```
def pure[A](a: A): Rand[A] =
  rng => (a, rng)
```

map

Transforms the output of a state action without modifying the state itself.

```
def map[A, B](s: Rand[A])(f: A => B): Rand[B] =
   rng => {
     val (a, rng2) = s(rng)
     (f(a), rng2)
   }
```

Example:

```
def nonNegativeEven: Rand[Int] =
   map(nonNegativeInt)(i => i - i % 2)
```


Combining State Actions: map2

- **map2** takes two actions, **ra** and **rb**, and a function **f** for combining their results, and returns a new action that combines them:

```
def map2[A,B,C](
  ra:Rand[A],
  rb:Rand[B]
)(f: (A, B) => C): Rand[C] = ???
```

Application of map2

```
def both[A,B](ra: Rand[A], rb: Rand[B]): Rand[(A,B)] =
  map2(ra, rb)((_, _))

val randIntDouble: Rand[(Int, Double)] =
  both(int, double)

val randDoubleInt: Rand[(Double, Int)] =
  both(double, int)
```


Exercise: *Hard*

EXERCISE 6.7

Hard: If you can combine two RNG transitions, you should be able to combine a whole list of them. Implement `sequence` for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the latter, you can use the standard library function `List.fill(n)(x)` to make a list with `x` repeated `n` times.

```
def sequence[A](fs: List[Rand[A]]): Rand[List[A]]
```

Nesting State Actions

```
def nonNegativeLessThan(n: Int): Rand[Int] =
  map(nonNegativeInt) { _ % n }
```

- The generated numbers may be skewed because **Int.MaxValue** may not be exactly divisible by **n**. Then, how to rectify this?

```
def nonNegativeLessThan(n: Int): Rand[Int] =
  map(nonNegativeInt) { i =>
    val mod = i % n
    if (i + (n-1) - mod >= 0) mod else nonNegativeLessThan(n)(???)
  }
```

Retry recursively if the `Int` we got is higher than the largest multiple of `n` that fits in a 32-bit `Int`.

Manual Passing instead of using "map"

```
def nonNegativeLessThan(n: Int): Rand[Int] =
  rng =>
    val (i, rng2) = nonNegativeInt(rng)
    val mod = i % n
    if i + (n-1) - mod >= 0 then
      (mod, rng2)
    else
      nonNegativeLessThan(n)(rng2)
```

Combinator: flatMap

- But it would be better to have a combinator that does this passing along for us. Neither **map** nor **map2** will cut it. We need a more powerful combinator, **flatMap**.

```
def flatMap[A, B](f: Rand[A])(g: A => Rand[B]): Rand[B]
  = ???
```

- Implement **flatMap**, and then use it to implement **nonNegativeLessThan**.
- Reimplement **map** and **map2** in terms of **flatMap**!

we say that flatMap is more *powerful* than map and map2.

A General State Action Data Type

unit, **map**, **map2**, **flatMap**, and **sequence** are general-purpose functions for working with **state actions**.

Don't care about the type of the state such as **Rand**.

Generalize **Rand** to a new type to handle any type:

```
type State[S,+A] = S => (A,S)    or
case class State[S,+A](run: S => (A,S))
```

We can now just make **Rand** a type alias for **State**:

```
type Rand[A] = State[S,A]
```

State Type defined as Case Class

```scala
case class State[S,+A](run: S => (A,S)) {
  def map[B](f: A => B): State[S, B]
  def map2[B,C](r: State[S, B])(f: (A, B) => C): State[S, C]
  def flatMap[B](f: A => State[S, B]): State[S, B]
}

object State {
  def pure[S, A](a: A): State[S, A]
  def sequence[S,A](sas: List[State[S, A]]): State[S, List[A]]
  …
}
```


Scala3 Only: Opaque Type

```scala
opaque type State[S, +A] = S => (A, S)

object State:
  …
  extension [S, A](underlying: State[S, A])
    def run(s: S): (A, S) = underlying(s)
    def apply[S, A](f: S => (A, S)): State[S, A] = f
```

*At the cost of some boilerplate conversions, we get both **encapsulation** and **performance**.*

- An opaque type behaves like a type alias inside the defining scope.
  - "Defining scope" refers to the object containing the definition, or if the definition is top level, the package containing it.
- Outside of the defining scope though, the opaque type is unrelated to the representation type.

Purely functional imperative programming

```
val ns: Rand[List[Int]] =
  int.flatMap(x =>
    int.flatMap(y =>
      ints(x).map(xs =>
        xs.map(_ % y))))
```

≡

```
val ns: Rand[List[Int]] =
  for
    x <- int
    y <- int
    xs <- ints(x)
  yield xs.map(_ % y)
```


Two Constructors for Imperative Style

```
// Passes the incoming state along and returns it as the value
def get[S]: State[S, S] = s => (s, s)

// Replaces the state and returns unit as the value
def set[S](s: S): State[S, Unit] = _ => ((), s)

// Modifies the incoming state by the function f
def modify[S](f: S => S): State[S, Unit] = for
  s <- get
  _ <- set(s)
yield ()
```