IT5100B

Industry Readiness
Stream Processing

LECTURE 2

Generics

FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

TYPES

WHY LEARN ABOUT TYPES?

```java
class MyCoolClass {
    AType a;
    AnotherType b;
    YetAnotherType myCoolMethod(YetAnotherOne c) {
        return c.getYetAnotherType();
    }
}
```

Java is statically-typed; **all types are known** at compile-time

Understanding Java's type system is **imperative**

TYPES

WHAT ARE TYPES?

- Logical representations of meaning
- A contract where all members of the same type have common characteristics

Types are everywhere:

- Every variable/attribute/method parameter has a type
- Every class/interface is a type
- Every object/value is of some type

TYPES

- Expressions like `"abc" + 2` produce an object/value which itself is of a type
- Logically, this expression represents/mean some string
- It behaves similarly to all other `String`s; it has the same attributes and methods as other `String`s

- Variables/attributes/methods have a fixed type throughout its lifetime
- Only objects of the same type can be assigned into it
- We don’t need to know the exact object stored in/produced from it but we will know all the operations we can do on it

TYPES OF TYPES

1. Primitive types (value types, therefore they are pass-by-value)

2. Reference types (objects, therefore they are pass-by-reference)

CONFLICTING TYPES

How do we explain the following:

```java
int i = 1; // same type on both sides of the equation
int j = i; // -||-
String s = i; // error, different type
double d = i; // ok, even though different type?
```

Polymorphism!

PRIMITIVE WIDENING

```
double
  ▲
float
  ▲
long
  ▲
 int
  ▲  ↖
short   char
  ▲
byte
```

```java
byte b = 1;
int i = b;
double d = i;
```

AUTOBOXING/UNBOXING

```java
Integer i = 1;
int j = i;
double d = i;
```

Primitive types can be boxed into / unboxed from their corresponding reference types

SUBTYPING

```java
class A { }
class B extends A { }
A a = new B();
```

SUBTYPING

AKA SUBTYPE POLYMORPHISM

When a subtype inherits (extends/implements) some supertype, subtype can be treated as a supertype because

- Logically, objects of subtype are also objects of supertype
- Subtype inherits all properties of supertype, so contract is not broken

If S extends/implements T, we say S is a subtype of T:

$S <: T$

S is-a T

S satisfies all properties that T does (and more)

$<:$ is reflexive ($A <: A$ for all A)

$<:$ is transitive ($A <: B, B <: C$ implies $A <: C$)

KEY POINT #1

A type S can be treated as another type T if

- S primitive widens to T or
- S boxes/unboxes to R and R can be treated as T or
- S is a subtype of T

GENERICS

```java
class Meal {
    AlaCarteBurger b;
    String size;
}
```

Continuing from our POS problem last week, a `Meal` is a pair of a `AlaCarteBurger` and a `size`

```java
class Fraction {
    int num;
    int denom;
}
```

More generally, we can create pairs that store two things; example, a pair of an `int` and another `int` (like representing `Fraction`s)

```java
class Pair {
    Object first;
    Object second;
}
```

One class for each pair to create is cumbersome; just create one `Pair` class that holds a pair of any objects!

GENERICS

```java
class Pair {
    Object first;
    Object second;
}
```

```
jshell> Pair meal = new Pair("BVPB", "L");
meal ==> Pair
jshell> Pair frac = new Pair(1, 2);
frac ==> Pair
jshell> int num = (int) frac.first;
num  ==> 1
```

- Elements of this `Pair` are all `Object`
- We can put anything into this `Pair`
- However, when retrieving elements, compiler only ascertains that the elements are `Objects`
- Requires typecasting if we want to assert that the retrieved element is `int`
- Typecasting is done at **runtime** and **prone to abuse**

GENERALIZING PROGRAMS

We "generalize" programs by retaining similarities and parameterizing differences

Parameterizing values

```python
# parameterizing length
# of triangle
def draw_triangle(n):
    for _ in range(3):
        fd(n)
        lt(120)
```

Parameterizing behaviour

```python
# parameterizing behaviour
# to obtain summed term
def sum_stuff(n, f):
    return sum(map(
        f,
        range(n)))
```

Parameterizing **types**

```java
// parameterizing types!
class Pair<T, U> {
    T first;
    U second;
}
```

Implementation is **independent** of arguments

```java
import java.util.function.BiFunction;
class Pair<T, U> {
    T first; U second;
    Pair(T t, U u) { first = t; second = u; }
    <R> R combine(BiFunction<T, U, R> f) {
        return f.apply(first, second);
    }
    @Override
    public String toString() {
        return String.format("(%s, %s)", first, second);
    }
}
```

```
jshell> Pair<Integer, Integer> myFrac = new Pair<>(3, 5);
myFrac ==> (3, 5)
jshell> myFrac.combine((x, y) -> (double) x / y)
$1     ==> 0.6
jshell> Pair<String, String> meal = new Pair<>("BVPB", "L");
meal   ==> (BVPB, L)
jshell> meal.combine((b, s) -> b + " " + s)
$2     ==> "BVPB L"
```

PARAMETRIC POLYMORPHISM

Generic classes/methods can become multiple types depending on type arguments supplied

When type arguments are obvious, they can be inferred by the compiler

KEY POINT #2

Types themselves can be **parameterized**

VARIANCE

MIGHT BE CONFUSING

```
class Clinic<T> {
    private T value;
    Clinic(T t) {
        value = t;
    }
    public void receive(T t) {
        value = t;
    }
    public T release() {
        return value;
    }
}
```

```
class Animal { }
class Dog extends Animal { }
```

For the sake of an example, consider the four classes where:

- `Clinic<T>` can produce a `T` and consume a `T`
- `Hospital<U>` is-a `Clinic<U>`
- A `Dog` is-a `Animal`

```
class Hospital<U> extends Clinic<U> {
    Hospital(U u) {
        super(u);
    }
}
```

NUS National University of Singapore | School of Computing

VARIANCE

ARE THESE LEGAL?

```
Clinic<Dog> b = new Hospital<Dog>(new Dog());
```

Yes it is legal because for all `U`, `Hospital<U>` <: `Clinic<U>`

```
Clinic<Animal> b = new Clinic<Dog>(new Shirt());
```

No it is not legal because `Clinic<Dog>` cannot consume `Animal`

```
Clinic<Dog> b = new Clinic<Animal>(new Clothes());
```

No it is not legal because `Clinic<Animal>` cannot produce a `Dog`

`Clinic<Animal>` and `Clinic<Dog>` are **invariant** because

- An animal clinic should be able to **receive** any animal, but a dog clinic cannot
- A dog clinic should be able to **produce** a dog specifically, but an animal clinic cannot guarantee that

```
class Clinic<T> {
    private T value;
    Clinic(T t) {
        value = t;
    }
    public void receive(T t) {
        value = t;
    }
    public T release() {
        return value;
    }
}
```

```
class Animal { }
class Dog extends Animal { }
```

```
class Hospital<U> extends Clinic<U> {
    Hospital(U u) {
        super(u);
    }
}
```

VARIANCE

INVARIANCE OF PARAMETERIZED TYPES

Therefore,

- If we need the clinic to receive **and** release any animal, we **must use** an animal clinic
- If we need the clinic to receive **and** release dogs, we **must use** a dog clinic
- Even though a dog is-a animal, a dog clinic **is not** an animal clinic

VARIANCE

COVARIANCE & CONTRAVARIANCE WITH WILDCARDS

What if we only need the clinic to be a **producer** of animals? What types of clinics allow that?

- `Clinic<Animal>`
- `Clinic<Dog>`
- `Clinic<S>` where `S` <: `Animal`

```
Clinic<? extends Animal> b =
        new Clinic<>(new Dog());
Animal c = b.release();
```

What if we only need the clinic to be a **consumer** of dogs? What types of boxes allow that?

- `Clinic<Dog>`
- `Clinic<Animal>`
- `Clinic<S>` where `S` :> `Dog`

```
Clinic<? super Dog> b =
        new Clinic<>(new Animal());
b.consume(new Dog());
```

VARIANCE

COVARIANCE & CONTRAVARIANCE WITH WILDCARDS

When you need a producer of `T`, use `? extends T`

When you need a consumer of `T`, use `? super T`

Rule of thumb:

Producer **E**xtends; **C**onsumer **S**uper (**PECS**)

If you just need some clinic, use `Clinic<?>`

KEY POINT #3

Parameterized types are **invariant**; if you need variance, use **[bounded] wildcards**

JAVA COLLECTIONS API

BETTER THAN ARRAYS

Interface	Commonly-Used Implementation
`java.util.List`	`java.util.ArrayList`
`java.util.Set`	`java.util.HashSet`
`java.util.Map`	`java.util.HashMap`

`HashSet` and `HashMap` are implemented using hash tables; objects to store (set elements or map keys) must correctly implement `public int hashCode()`

JAVA FUNCTIONS API

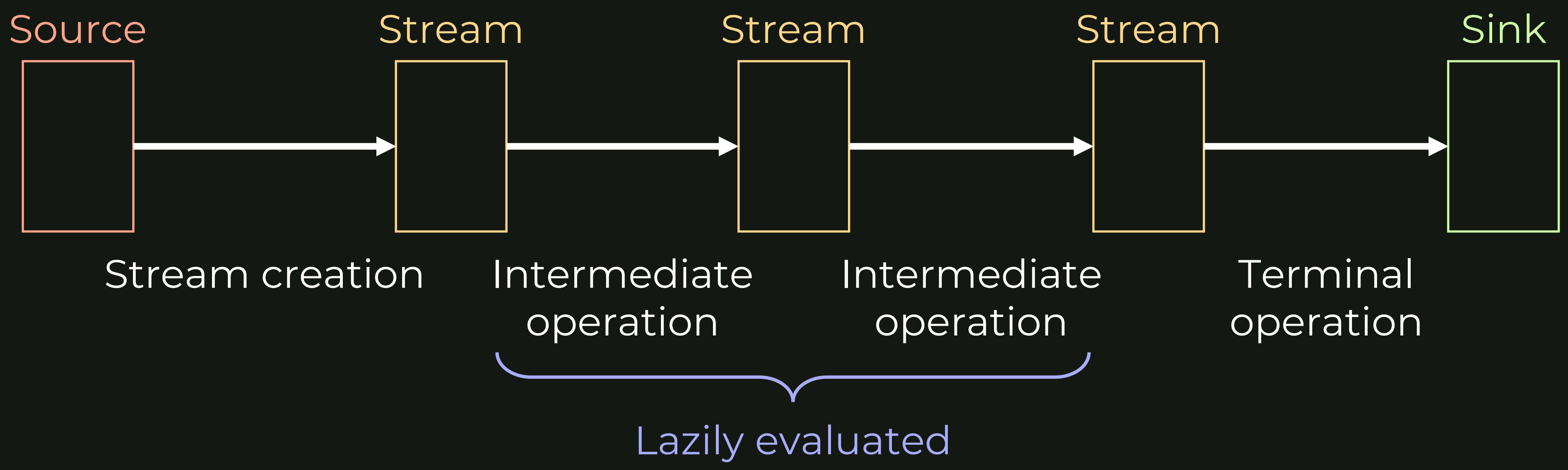
```
jshell> @FunctionalInterface
   ...> public interface MyCoolFunction<T, U> {
   ...>     U beCool(T t);
   ...> }
|  created interface MyCoolFunction
jshell> MyCoolFunction<Integer, String> f = x -> "A" + x;
f ==> $Lambda$17/0x0000000840078440@46f7f36a
jshell> f.beCool(10)
$1 ==> "A10"
```

- Functional Interfaces are interfaces with a single abstract method
- Lambda expressions are expanded to instances of functional interfaces
- The Functions API contains many frequently used functional interfaces

STREAMS AND OPTIONALS

STREAMS

STREAM PIPELINE



Collections do not support declarative statements

Use Streams (lazily-evaluated potentially infinitely large single-use iterables)

[`java.util.stream`]

NUS National University of Singapore | School of Computing

STREAMS

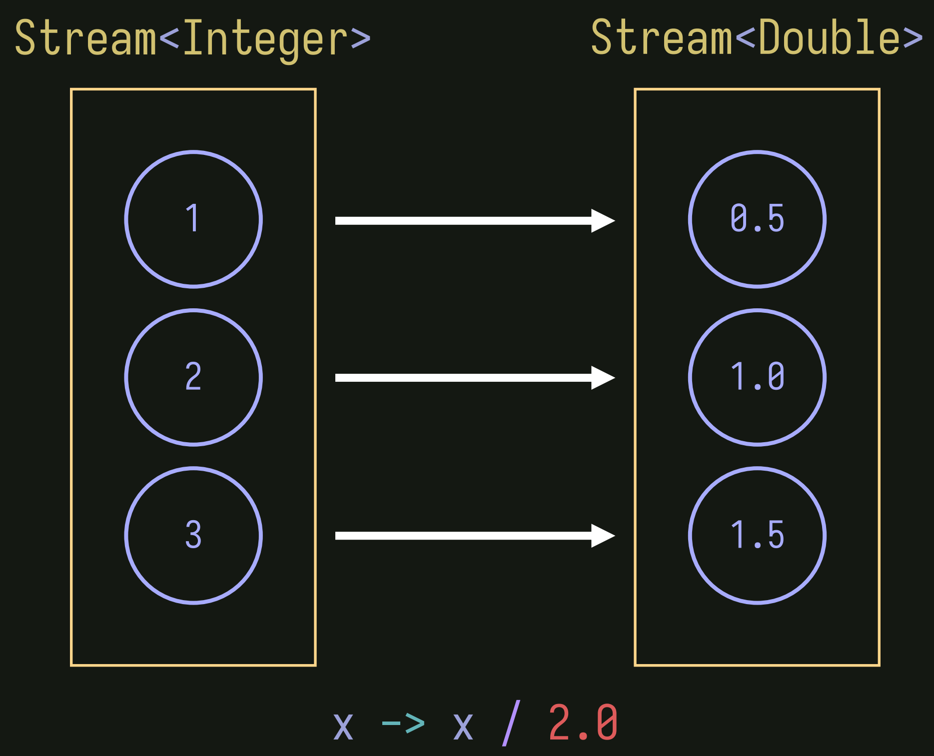
CREATING STREAMS

```
jshell> List<Integer> ls = List.of(1, 2, 3, 4);
jshell> Stream<Integer> myStream = ls.stream();
jshell> myStream = Stream.of(1, 2, 3, 4);
jshell> Integer[] arr = new Integer[]{1, 2, 3, 4};
jshell> myStream = Arrays.stream(arr);
jshell> myStream = Stream.generate(() -> 1);
jshell> myStream = Stream.iterate(1, x -> x + 1);c
```

Several ways to create a stream from other collections or from functions

STREAMS

INTERMEDIATE OPERATIONS

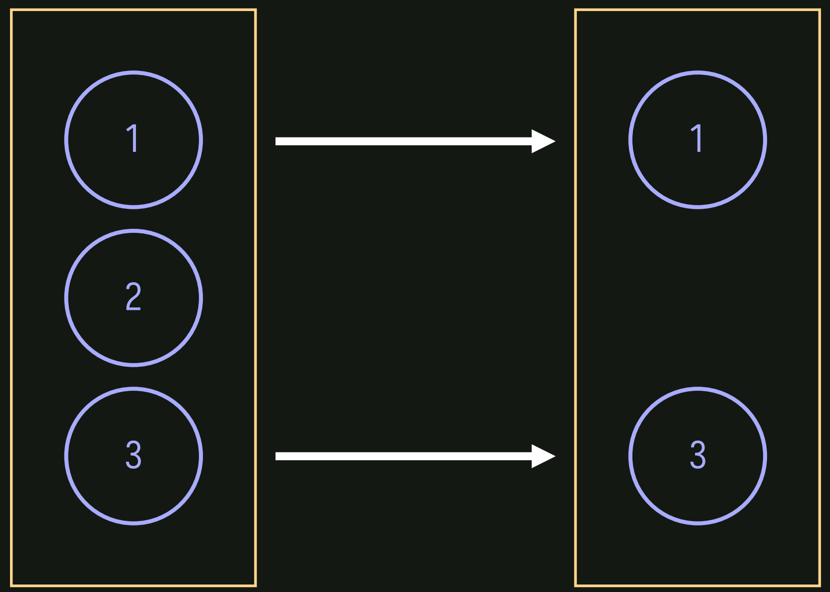


Map: transform each element of the stream using a function

STREAMS

INTERMEDIATE OPERATIONS

Stream<Integer> Stream<Integer>

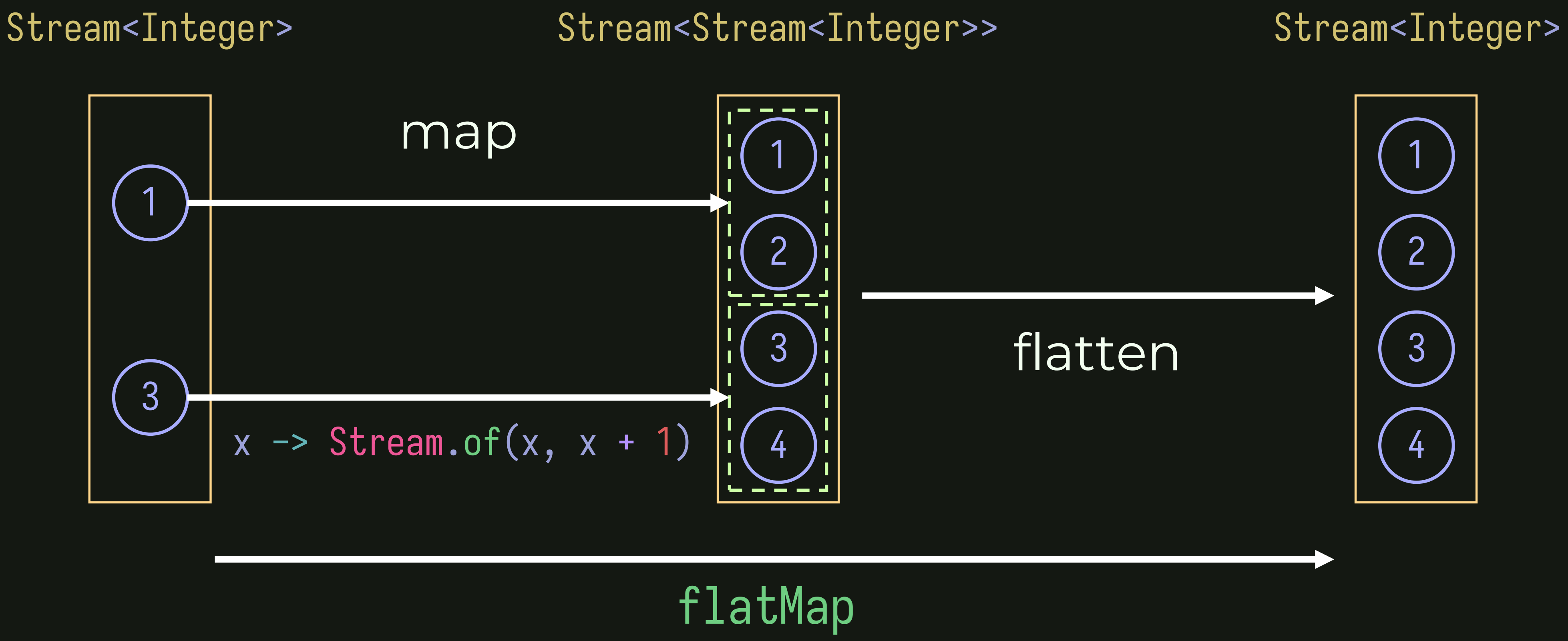



```
x -> x % 2 != 0
```

Filter: new stream with elements satisfying predicate

STREAMS

INTERMEDIATE OPERATIONS



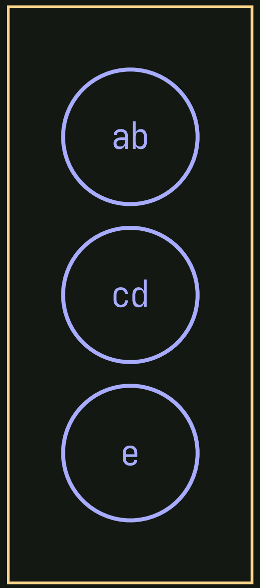
flatMap: transform each element of the stream into a stream using a function, then flatten the whole stream

NUS National University of Singapore | School of Computing

STREAMS

TERMINAL OPERATIONS

`Stream<String>`




```
f = (x, y) -> x + " " + y
```

```
f(f("ab", "cd"), "e") // "ab cd e"
```

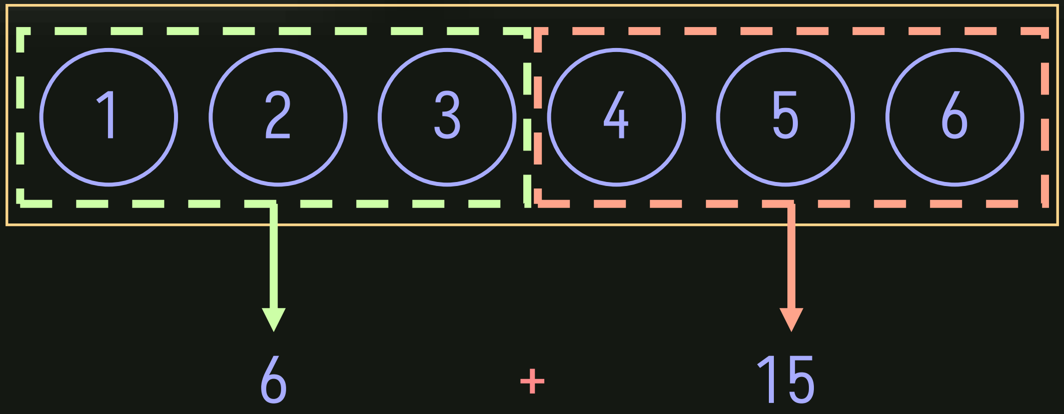
Reduce: left-associative pairwise reduction over a stream with optional initial value

STREAMS

SEQUENTIAL VS PARALLEL

```
for (int i : numbers) {
        total += i;
}
```

```
numbers.parallel()
        .reduce(0, (x, y) -> x + y)
```



Streams support **parallel processing**: distributing workload to several process threads to be done in **parallel**

STREAMS

CONDITIONS FOR PARALLELISM

Ideally, only parallelize operations that satisfy:

- Non-interference (do not amend source during stream operation)
- Statelessness (avoid stateful computation)
- No side-effects (avoid producing side-effects)
- Associativity (`(a op b) op c == a op (b op c)`)

OPTIONALS

```java
static String get(int i) {
  if (i == 1)
    return "Hello!";
  return null;
}
String s = get(2);
System.out.println(s.length());
// NullPointerException
```

```java
static Optional<String> get(int i) {
  if (i == 1)
    return Optional.of("hello");
  return Optional.empty();
}
Optional<String> s = get(2);
if (s.isPresent())
  System.out.println(s.get().length());
```

Instead of using `null`, use `Optional`s which represent potentially empty values

Force users to acknowledge and handle empty case

OPTIONALS

WORKING WITH OPTIONALS

```java
static Function<Integer, Optional<Integer>> divideBy(int i) {
  return x -> {
      if (i == 0)
          return Optional.empty();
      return Optional.of(x / i);
  };
}
```

```java
Optional<Integer> o = divideBy(4).apply(9);
if (o.isPresent()) {
  int i = o.get();
  o = divideBy(2).apply(i);
  if (o.isPresent()) {
    i = o.get();
    System.out.println(i);
  }
}
```

```java
Optional.of(9)
  .flatMap(divideBy(4))
  .flatMap(divideBy(2))
  .ifPresent(System.out::println);
```

Use declarative statements to work with `Optional`s easily!

KEY POINT #4

Read up on the APIs and start working with different generic types declaratively

WORKED EXAMPLE

Let's create our own `ImmutableList` type which can

- Be prepended
- Be concatenated
- Be reduced
- Get the first element
- Get the slice of the list excluding the first element
- … (other typical list methods)

Also create a factory method that receives a bunch of objects and puts them in an `ImmutableList`

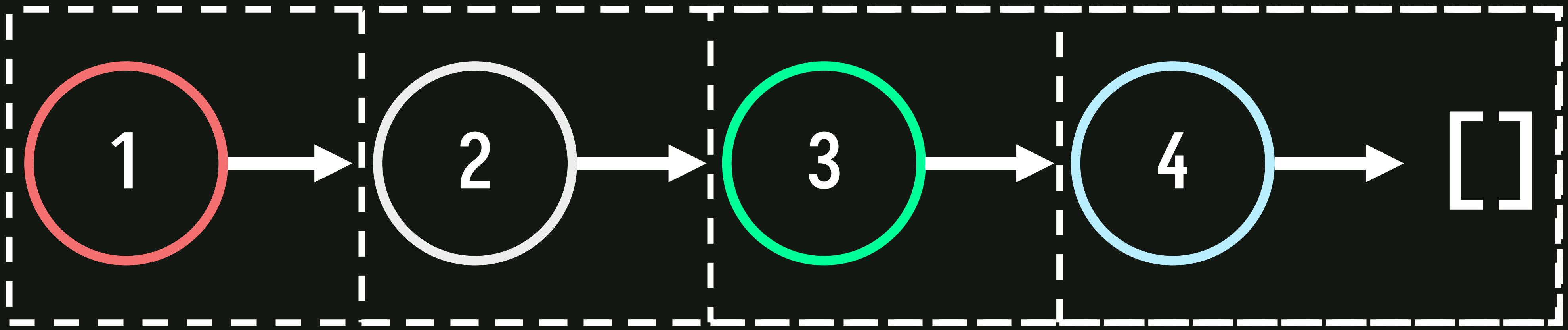
WORKED EXAMPLE

IMMUTABLELIST INTERFACE

```java
interface ImmutableList<T> {
    ImmutableList<T> prepended(T t);
    ImmutableList<T> concat(ImmutableList<T> other);
    T reduce(BiFunction<T, T, T> f);
    T head();
    ImmutableList<T> tail();
    @SuppressWarnings("unchecked")
    static <T> ImmutableList<T> of(T... values) {
        // ...
    }
}
```

WORKED EXAMPLE

IMMUTABLELIST IMPLEMENTATION



A list is either:

1) A node with a head value and a reference to the tail
2) An empty list

```java
class ImmutableListNode<T> implements ImmutableList<T> {
    private final T h;
    private final ImmutableList<T> t;
    ImmutableListNode(T head, ImmutableList<T> tail) {
        h = head;
        t = tail;
    }
    public T head() { return h; }
    public ImmutableList<T> tail() { return t; }
    public boolean isEmpty() { return false; }
    public ImmutableList<T> prepended(T t) {
        return new ImmutableListNode<T>(t, this);
    }
    public ImmutableList<T> concat(ImmutableList<T> other) {
        return t.concat(other).prepended(h);
    }
    public T reduce(BiFunction<T, T, T> f) {
        if (t.isEmpty()) return h;
        return t.tail().prepended(f.apply(h, t.head())).reduce(f);
    }
    @Override
    public String toString() {
        if (t.isEmpty()) return h.toString();
        return String.format("%s : %s", h, t);
    }
}
```

WORKED EXAMPLE

IMMUTABLELISTNODE

You may also add other convenience methods

WORKED EXAMPLE

IMMUTABLEEMPTYLIST

Effectively just write all the base cases here

```java
class ImmutableEmptyList<T> implements ImmutableList<T> {
    public T head() { return null; }
    public ImmutableList<T> tail() { return null; }
    public T reduce(BiFunction<T, T, T> f) { return null; }
    public boolean isEmpty() { return true; }
    public ImmutableList<T> prepended(T t) { return new ImmutableListNode<T>(t, this); }
    public ImmutableList<T> concat(ImmutableList<T> other) { return other; }
    @Override
    public String toString() { return ""; }
}
```

CONTENTS

- Types
- Streams and Optionals

KEY POINTS

- Types themselves can be **parameterized**
- Parameterized types are **invariant**; if you need variance, use **[bounded] wildcards**
- Read up on the APIs and start working with different generic types declaratively