

IT5100B

Industry Readiness
Stream Processing

LECTURE 2

Generics

FOO Yong Qi
yongqi@nus.edu.sg

CONTENTS

- Types
- Streams and Optionals

TYPES

WHY LEARN ABOUT TYPES?

```
1 | class MyCoolClass {  
2 |     AType a;  
3 |     AnotherType b;  
4 |     YetAnotherType myCoolMethod(YetAnotherOne c) {  
5 |         return c.getYetAnotherType();  
6 |     }  
7 | }
```

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 Strings; it has the same attributes and methods as other Strings
- 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:

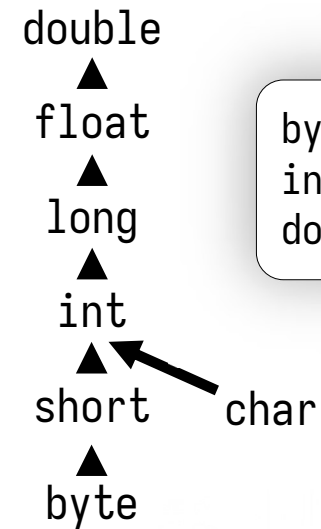
```
int i = 1; // same type on both sides of the equation
int j = i; // -||-
String s = i; // error, different type
double d = i; // ok, even though different type?
```

Polymorphism!

PRIMITIVE WIDENING

AUTOBOXING/UNBOXING

SUBTYPING



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

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

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

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

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

GENERICIS

```
1 class Meal {  
2     AlaCarteBurger b;  
3     String size;  
4 }
```

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

```
1 class Fraction {  
2     int num;  
3     int denom;  
4 }
```

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

```
1 class Pair {  
2     Object first;  
3     Object second;  
4 }
```

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

GENERIC

```
1 class Pair {  
2     Object first;  
3     Object second;  
4 }
```

```
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

```
1 | # parameterizing length
2 | # of triangle
3 | def draw_triangle(n):
4 |     for _ in range(3):
5 |         fd(n)
6 |         lt(120)
```

Parameterizing behaviour

```
1 | # parameterizing behaviour
2 | # to obtain summed term
3 | def sum_stuff(n, f):
4 |     return sum(map(
5 |         f,
6 |         range(n)))
```

Parameterizing **types**

```
1 | // parameterizing types!
2 | class Pair<T, U> {
3 |     T first;
4 |     U second;
5 | }
```

Implementation is **independent** of arguments

```

1 import java.util.function.BiFunction;
2 class Pair<T, U> {
3     T first; U second;
4     Pair(T t, U u) { first = t; second = u; }
5     <R> R combine(BiFunction<T, U, R> f) {
6         return f.apply(first, second);
7     }
8     @Override
9     public String toString() {
10         return String.format("(%s, %s)", first, second);
11     }
12 }

```

```

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

```
1 class Clinic<T> {  
2     private T value;  
3     Clinic(T t) {  
4         value = t;  
5     }  
6     public void receive(T t) {  
7         value = t;  
8     }  
9     public T release() {  
10        return value;  
11    }  
12 }
```

```
1 class Animal { }  
2 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

```
1 class Hospital<U> extends Clinic<U> {  
2     Hospital(U u) {  
3         super(u);  
4     }  
5 }
```

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

```
1 class Clinic<T> {  
2     private T value;  
3     Clinic(T t) {  
4         value = t;  
5     }  
6     public void receive(T t) {  
7         value = t;  
8     }  
9     public T release() {  
10        return value;  
11    }  
12 }
```

```
1 class Animal { }  
2 class Dog extends Animal { }
```

```
1 class Hospital<U> extends Clinic<U> {  
2     Hospital(U u) {  
3         super(u);  
4     }  
5 }
```


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
<u>java.util.List</u>	<u>java.util.ArrayList</u>
<u>java.util.Set</u>	<u>java.util.HashSet</u>
<u>java.util.Map</u>	<u>java.util.HashMap</u>

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

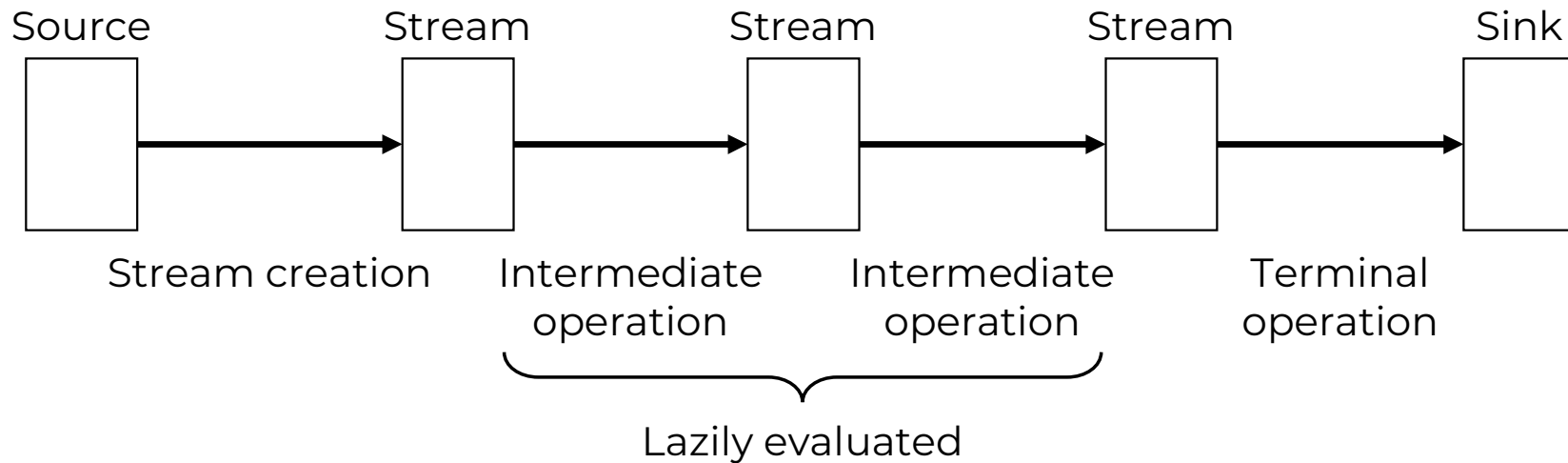
```
jshell> @FunctionalInterface
...> public interface MyCoolFunction<T, U> {
...>     U beCool(T t);
...> }
| created interface MyCoolFunction
jshell> MyCoolFunction<Integer, String> f = x -> "A" + x;
f ==> $Lambda$17/0x0000000084007844@46f7f36a
jshell> f.beCool(10)
$1 ==> "A10"
```

- 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`

STREAMS

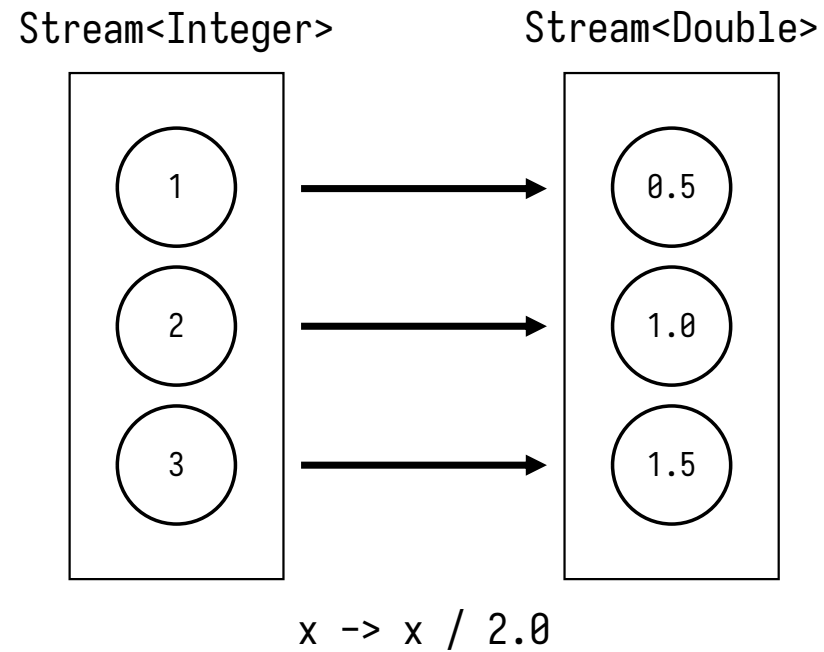
CREATING STREAMS

```
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

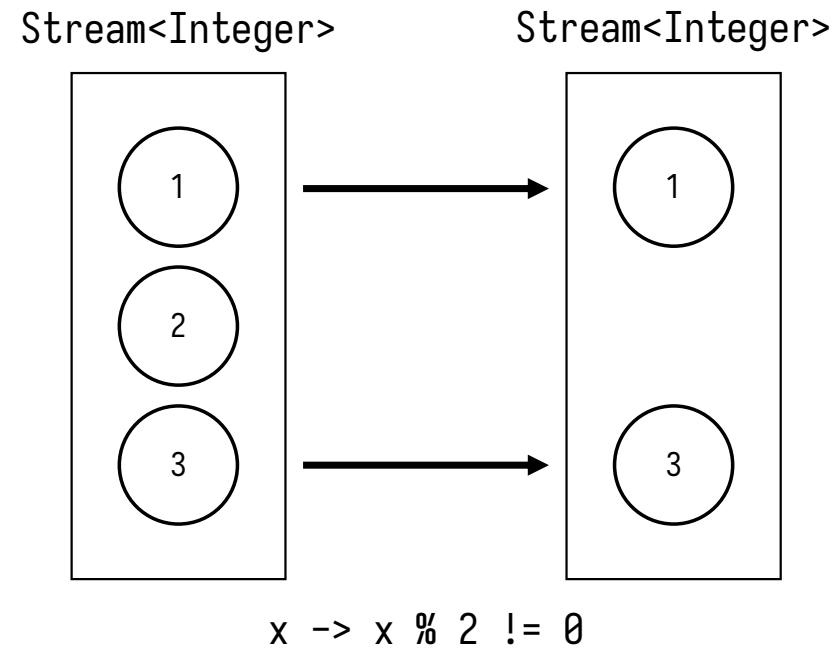
INTERMEDIATE OPERATIONS



Map: transform each element of the stream using a function

STREAMS

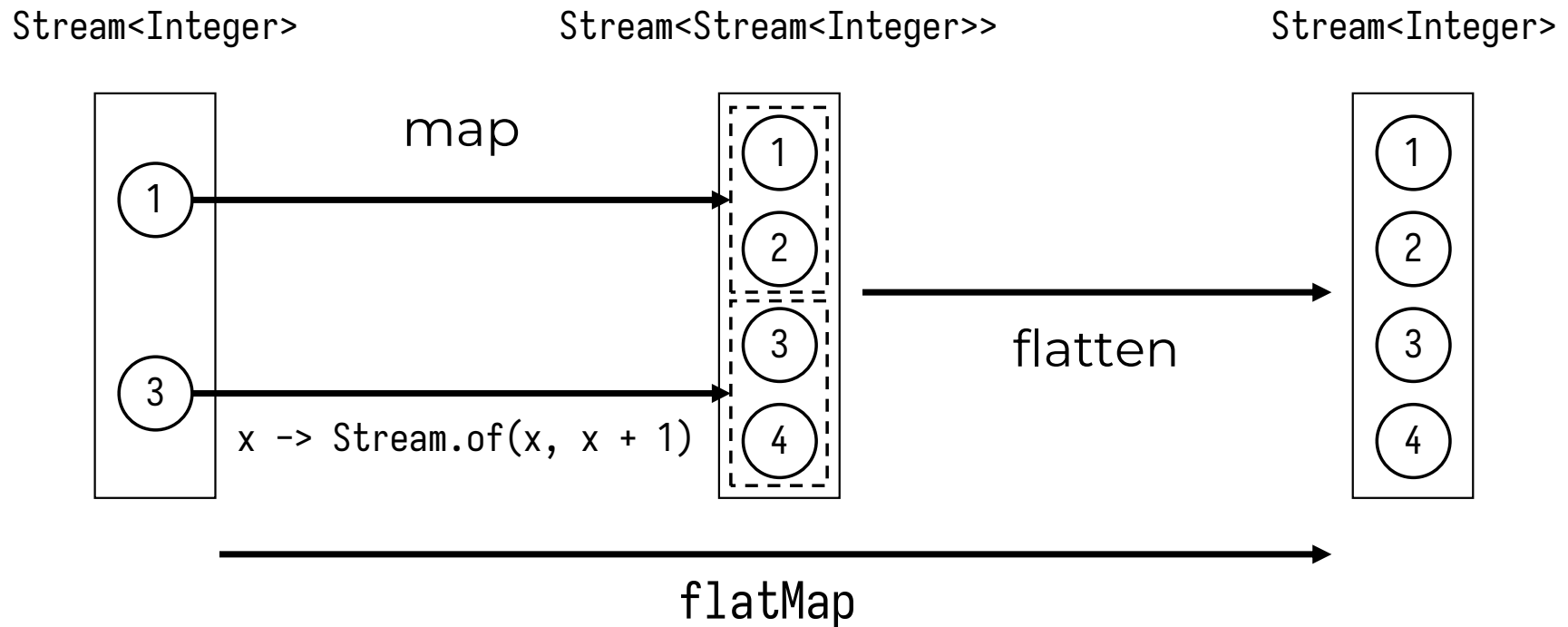
INTERMEDIATE OPERATIONS



Filter: new stream with elements satisfying predicate

STREAMS

INTERMEDIATE OPERATIONS

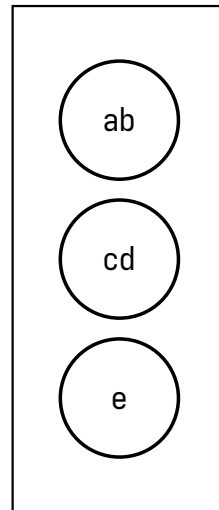


`flatMap`: transform each element of the stream into a stream using a function, then flatten the whole stream

STREAMS

TERMINAL OPERATIONS

Stream<String>



$f = (x, y) \rightarrow x + " " + y$

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

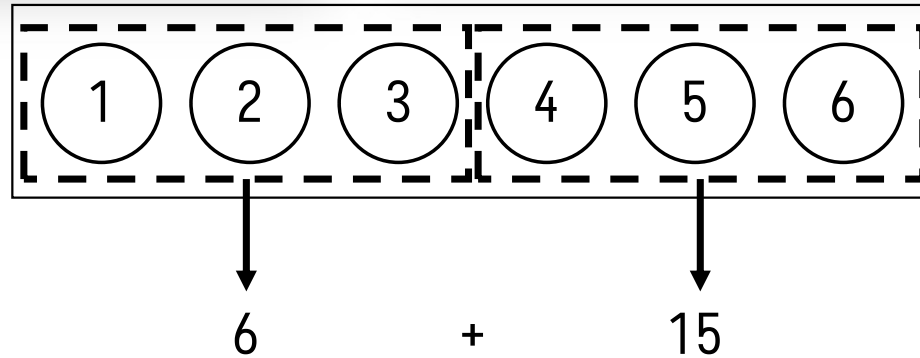
Reduce: left-associative pairwise reduction over a stream with optional initial value

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 $((a \text{ op } b) \text{ op } c == a \text{ op } (b \text{ op } c))$

OPTIONALS

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

```
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 `Optionals` which represent potentially empty values

Force users to acknowledge and handle empty case

OPTIONALS

WORKING WITH OPTIONALS

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

```
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);  
    }  
}
```

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

Use declarative statements to work with Optionals 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`

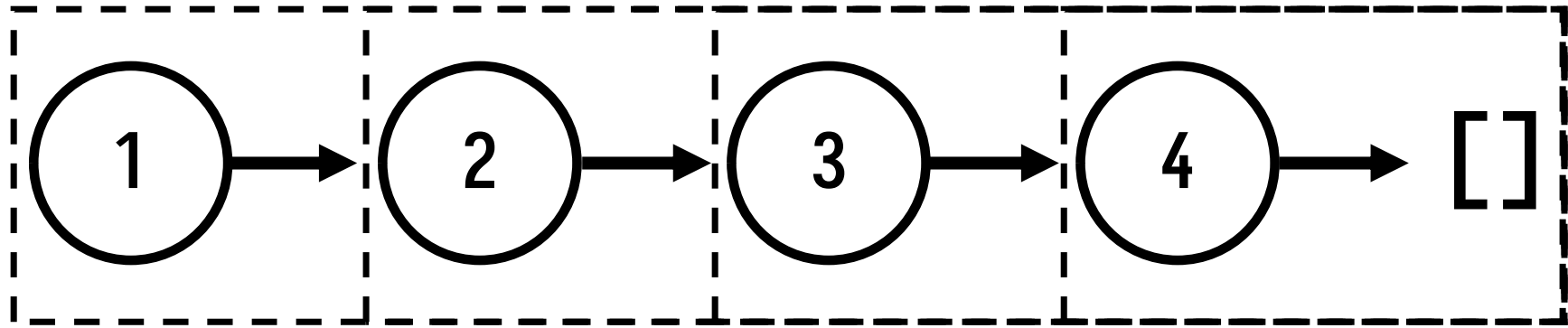
WORKED EXAMPLE

IMMUTABLELIST INTERFACE

```
1 interface ImmutableList<T> {  
2     ImmutableList<T> prepended(T t);  
3     ImmutableList<T> concat(ImmutableList<T> other);  
4     T reduce(BiFunction<T, T, T> f);  
5     T head();  
6     ImmutableList<T> tail();  
7     @SuppressWarnings("unchecked")  
8     static <T> ImmutableList<T> of(T... values) {  
9         // ...  
10    }  
11 }
```

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

```

1 class ImmutableListNode<T> implements ImmutableList<T> {
2     private final T h;
3     private final ImmutableList<T> t;
4     ImmutableListNode(T head, ImmutableList<T> tail) {
5         h = head;
6         t = tail;
7     }
8     public T head() { return h; }
9     public ImmutableList<T> tail() { return t; }
10    public boolean isEmpty() { return false; }
11    public ImmutableList<T> prepended(T t) {
12        return new ImmutableListNode<T>(t, this);
13    }
14    public ImmutableList<T> concat(ImmutableList<T> other) {
15        return t.concat(other).prepending(h);
16    }
17    public T reduce(BiFunction<T, T, T> f) {
18        if (t.isEmpty()) return h;
19        return t.tail().prepending(f.apply(h, t.head())).reduce(f);
20    }
21    @Override
22    public String toString() {
23        if (t.isEmpty()) return h.toString();
24        return String.format("%s : %s", h, t);
25    }
26 }

```

WORKED EXAMPLE

IMMUTABLELISTNODE

You may also add other convenience methods

WORKED EXAMPLE

IMMUTABLEEMPTYLIST

Effectively just write all the base cases
here

```
1 class ImmutableList<T> implements ImmutableList<T> {  
2     public T head() { return null; }  
3     public ImmutableList<T> tail() { return null; }  
4     public T reduce(BiFunction<T, T, T> f) { return null; }  
5     public boolean isEmpty() { return true; }  
6     public ImmutableList<T> prepended(T t) { return new ImmutableListNode<T>(t, this); }  
7     public ImmutableList<T> concat(ImmutableList<T> other) { return other; }  
8     @Override  
9     public String toString() { return ""; }  
10 }
```

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