Optional Types

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 9

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
- `max()` of empty stream is undefined
  - Return value could be `Double` or `null`

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Dealing with empty streams

- Largest and smallest values seen
  - `max()` and `min()`
  - Requires a comparison function
  - What happens if the stream is empty?
- `max()` of empty stream is undefined
  - Return value could be `Double` or `null`
- `Optional<T>` object
  - Wrapper
  - May contain an object of type `T`
    - Value is present
  - Or no object

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);
```

Handling missing optional values

- Use `orElse()` to pass a default value

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand = maxrand.orElse(-1.0);
```

Handling missing optional values

- Use `orElse()` to pass a default value
- Use `orElseGet()` to call a function to generate replacement for a missing value

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand = maxrand.orElseGet(
     () -> SomeFunctionToGenerateDouble
   );
```

Handling missing optional values

- Use `orElse()` to pass a default value
- Use `orElseGet()` to call a function to generate replacement for a missing value
- Use `orElseThrow()` to generate an exception when a missing value is encountered

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Double fixrand =
   maxrand.orElseThrow(
     IllegalStateException::new
   );
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored

```
optionalValue.ifPresent(v -> Process v);
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresent(v -> results.add(v));
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present
  - As usual, pass the function in different forms

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresent(results::add);
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
  - Missing value is ignored
- For instance, add `maxrand` to a collection `results`, if it is present
  - As usual, pass the function in different forms
- Specify an alternative action if the value is not present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.ifPresentOrElse(
   v -> results.add(v),
   () -> System.out.println("No max")
);
```

Creating an optional value

- Creating an optional value
  - `Optional.of(v)` creates value `v`
  - `Optional.empty` creates empty optional

```
public static Optional<Double>
   inverse(Double x){

   if (x == 0) {
     return Optional.empty();
   }else{
     return Optional.of(1 / x);
   }
}
```

Creating an optional value

- Creating an optional value
  - `Optional.of(v)` creates value `v`
  - `Optional.empty` creates empty optional
- Use `ofNullable()` to transform `null` automatically into an empty optional
  - Useful when working with functions that return object of type `T` or `null`, rather than `Optional<T>`

```
public static Optional<Double>
   inverse(Double x) {

   return Optional.ofNullable(1 / x);
}
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> maxrandsqr =
  maxrand.map(v -> v*v);
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output
- Another example

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.map(results::add);
```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
  - If input is empty, so is output
- Another example
- Supply an alternative for a missing value
  - If value is present, it is passed as is
  - If value is empty, value generated by `or()` is passed

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> fixrand =
   maxrand.or(() -> Optional.of(-1.0));
```

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`
- Instead, use `flatMap`
  - `s.f().flatMap(T::g)`
  - If `s.f()` is present, apply `g()`
  - Otherwise return empty `Optional<U>`

```
Optional<U> result = s.f().flatMap(T::g);
```

Composing optional values of different types

- Suppose that
  - `f()` returns `Optional<T>`
  - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
  - `s.f()` has type `Optional<T>`, not `T`
- Instead, use `flatMap`
  - `s.f().flatMap(T::g)`
  - If `s.f()` is present, apply `g()`
  - Otherwise return empty `Optional<U>`
- For example, pass output of earlier safe `inverse()` to safe `squareRoot()`

```
public static Optional<Double>
   inverse(Double x) {
   if (x == 0) {
     return Optional.empty();
   }else{
     return Optional.of(1 / x);
   }
}
public static Optional<Double>
   squareRoot(Double x){
   if (x < 0) {
     return Optional.empty();
   }else{
     return Optional.of(Math.sqrt(x));
   }
}

Optional<Double> result =
  inverse(x).flatMap(MyClass::squareRoot);
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username

```
Optional<User> lookup(String id) {...}
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`

```
Optional<User> lookup(String id) {...}
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`
- Pass through a `flatMap`

```
Stream<String> ids = ...;
Stream<User> users = ids.map(Users::lookup)
   .flatMap(Optional::stream);
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username
- Want to convert a stream of userids into a stream of users
  - Input is `Stream<String>`
  - Output is `Stream<User>`
  - But `lookup` returns `Optional<User>`
- Pass through a `flatMap`
- What if `lookup` was implemented without using `Optional`?
  - `oldLookup` returns `User` or `null`
  - Use `ofNullable` to regenerate `Optional<User>`

```
Stream<String> ids = ...;
Stream<User> users = ids.flatMap(
   id -> Stream.ofNullable(
           Users.oldLookup(id)
         )
);
```

Summary

- `Optional<T>` is a clean way to encapsulate a value that may be absent
- Different ways to process values of type `Optional<T>`
  - Replace the missing value by a default
  - Ignore missing values
- Can create values of type `Optional<T>` where outcome may be undefined
- Can write functions that transform optional values to optional values
- `flatMap` allows us to cascade functions with optional types
  - Use `flatMap` to regenerate a stream from optional values