CS2030 Lecture 8

Programming with Contexts

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2023 / 2024

Lecture Outline and Learning Outcomes

- □ Understand the concept of a *computation context*
- □ Be able to define a computation context
  - – e.g. `Maybe` context to handle **`null`** values
- □ Know the difference between imperative and declarative styles of programming
- □ Understand how *higher order functions* can be used to support **cross-barrier manipulation**
- □ Appreciate `map` versus `flatMap`
- □ Awareness of *variable capture* associated with a *local class*
- □ Understand variable capture using the Java memory model

Computation Context

- □ A *computation context* wraps around a value, and abstracts away computations associated with the context
  - – a "safe box" in which functions can be safely executed
  - – e.g. `Optional` is a computation context that handles invalid or missing values
- □ A computation context comprises:
  - – a way to wrap the parameter within the box, e.g. using `of`

    `Optional<Integer> oi = Optional.<Integer>of(1)`
  - – a way to pass a behaviour into the box via a *higher order method* (method that takes in another method) so that it can be applied to the parameter value

Defining a *Maybe* Context

```
class Maybe<T> {
    private final T value;

    private Maybe(T value) { // declared private
        this.value = value;
    }

    static <T> Maybe<T> of(T value) { // generic method of type T that is
        if (value == null) {          // declared with method scope
            return Maybe.<T>empty();
        }
        return new Maybe<T>(value);
    }

    static <T> Maybe<T> empty() {
        return new Maybe<T>(null);
    }

    @Override
    public String toString() {
        if (this.value == null) {
            return "Maybe.empty";
        } else {
            return "Maybe[" + value + "]";
        }
    }
```

`get`, `isEmpty` and `isPresent` Methods

- Declared as private helper methods

```
private T get() {
    return value;
}

private boolean isEmpty() {
    return this.get() == null;
}

private boolean isPresent() {
    return !this.isEmpty();
}
```

- Prevents `Maybe` context being used imperatively
- Programming with contexts should be **declarative**
  - *declarative* programming specifies *what to do*
  - *imperative* programming specifies *how to do* a task

Overriding `equals` Method in `Maybe`

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof Maybe<?> other) { // note Maybe<?>
        return (this.isEmpty() && other.isEmpty()) ||
            (this.isPresent() && other.isPresent() &&
             this.value.equals(other.value));
    } else {
        return false;
    }
}
```

- `Maybe<?> other` can reference a `Maybe` of *any* type
- `this.get().equals(other.get())` is valid because
  - any object wrapped in `Maybe` has an `equals` method
  - any object wrapped in `Maybe` can be passed as an argument to an `equals` method

Cross–Barrier Manipulation

- **Cross-barrier manipulation** — where the client defines a function that is passed to the context for execution, e.g.
  - `Optional<T>::filter(Predicate<? super T>) : Optional<T>`

```
jshell> Predicate<Integer> pred = x -> x % 2 == 0
pred ==> $Lambda$20/0x00007f48d0009a08@27973e9b

jshell> Optional.<Integer>of(1).filter(pred)
$.. ==> Optional.empty

jshell> Optional.<Integer>of(2).filter(pred)
$.. ==> Optional[2]

jshell> Predicate<Object> pred = x -> x.equals(1)
p ==> $Lambda$21/0x00007f48d000a410@506e1b77

jshell> Optional.<Integer>of(1).filter(pred)
$.. ==> Optional[1]

jshell> Optional.<Integer>of(2).filter(pred)
$.. ==> Optional.empty

jshell> Optional.<Integer>empty().filter(pred)
$.. ==> Optional.empty
```

$$\texttt{Optional<Integer>} \xrightarrow{\texttt{filter(pred)}} \texttt{Optional<Integer>}$$

Conditional Expression

- A conditional expression comprises a **conditional operator** that is used in place of **`if`/`else`** construct
- It comprises three parts:
  - a condition that evaluates to **`true`** or **`false`**
  - an expression to perform if the condition is true
  - an expression to perform if the condition is false
- E.g. returning a conditional expression within a method

```
return a < b ? b - a : b + a;
```

is equivalent to

```
if (a < b) {
    return b - a;
} else {
    return b + a;
}
```

filter Method

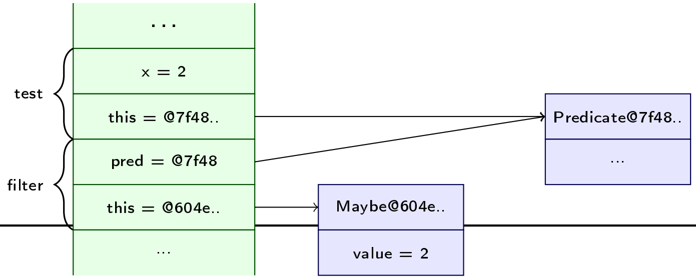
□ Define the following `filter` method in the `Maybe` class

```
Maybe<T> filter(Predicate<? super T> pred) {
    if (this.isPresent() && pred.test(this.get())) {
        return this;
    }
    return Maybe.<T>empty();
//  return this.isPresent() && pred.test(this.get()) ? this : Maybe.<T>empty();
}
```

```
jshell> Predicate<Integer> pred = x -> x % 2 == 0
pred ==> $Lambda$20/0x00007f48d0009a08@27973e9b

jshell> Maybe.<Integer>empty().filter(pred)
$.. ==> Optional.empty

jshell> Maybe.<Integer>of(2).filter(pred)
$.. ==> Optional[2]
```



ifPresent and map Methods

□ `ifPresent` takes in `Consumer<? super T>`; returns `void`

```
void ifPresent(Consumer<? super T> action) {
    if (this.isPresent()) {
        action.accept(this.get());
    }
}
```

```
jshell> Maybe.<Integer>empty().ifPresent(x -> System.out.println(x))

jshell> Maybe.<Integer>of(123).ifPresent(x -> System.out.println(x))
123
```

□ `map` takes in `Function<? super T, ? extends R>`; returns `Maybe<R>`

```
// declaration of <R> with method scope
<R> Maybe<R> map(Function<? super T, ? extends R> mapper) {
    return this.isEmpty() ? Maybe.<R>empty() :
        Maybe.<R>of(mapper.apply(this.get()));
}
```

```
jshell> Maybe.<Integer>empty().map(x -> x + 1)
$.. ==> Maybe.empty

jshell> Maybe.<Integer>of(123).map(x -> x + 1)
$.. ==> Maybe[124]
```

□ Mapping comes in two variants: `map` and `flatMap`

Java Optional's map versus flatMap

□ Using `map` with a function that results in an `Integer`

```
jshell> Function<Integer, Integer> f = x -> x + 1
f ==> $Lambda$20/0x00007f114000a618@4fca772d

jshell> Optional.of(2).map(f)
$.. ==> Optional[3]
```

$$\texttt{Optional<Integer>} \xrightarrow{\texttt{map(f:Integer} \to \texttt{Integer)}} \texttt{Optional<Integer>}$$

□ Using `map` with a function that results in an `Optional<Integer>`

```
jshell> Function<Integer, Optional<Integer>> g = x -> Optional.of(x + 1)
g ==> $Lambda$21/0x00007f114000ac68@133314b

jshell> g = x -> Optional.of(x).map(y -> y + 1) // alternatively
g ==> $Lambda$24/0x00007f114000c410@17a7cec2

jshell> Optional.of(2).map(g)
$.. ==> Optional[Optional[3]]
```

$$\texttt{Optional<Integer>} \xrightarrow{\texttt{map(f:Integer} \to \texttt{Optional<Integer>)}} \texttt{Optional<Optional<Integer>>}$$

□ Need to flatten the resulting context using `flatMap`

```
jshell> Optional.of(2).flatMap(g)
$.. ==> Optional[3]
```

Local Class and Variable Capture

□ Local class is declared locally within a code block

– anonymous inner class or lambda

□ Consider the anonymous inner class defined within class `A`

```
jshell> class A {
   ...>     private final int z;
   ...>     A(int z) { this.z = z; }
   ...>     Predicate<Integer> foo(int y) {
   ...>         return new Predicate<Integer>() {
   ...>             @Override
   ...>             public boolean test(Integer x) {
   ...>                 return x == y + z; // or return x == y + A.this.z;
   ...>             }
   ...>         };
   ...>     }
   ...> }
|  created class A
```

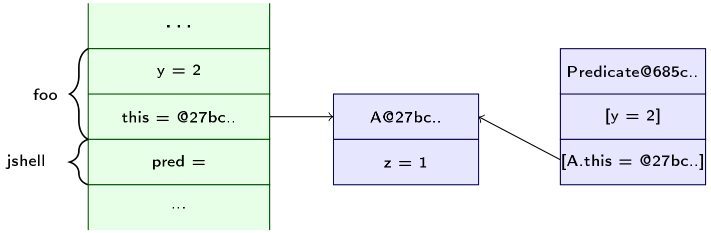
□ *Variable capture*: local class makes a copy of variables of the enclosing method and reference to the enclosing class

Java Memory Model

- □ Memory model of the statement

```
jshell> Predicate<Integer> pred = new A(1).foo(2)
pred ==> A$1@27bc2616
```

just before returning from the method `foo`

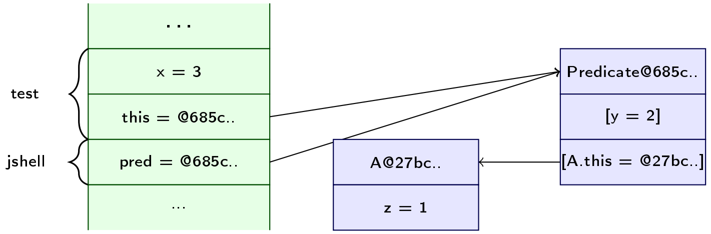


- □ *Closure*: local class closes over it's enclosing method and class
  - – local variables of the method (e.g. `y`) are captured
  - – reference of the enclosing class (e.g. `A.this`)* is captured

*(`A.this`) is called a *qualified this*

Java Memory Model

- □ Memory model upon invoking the method `pred.test(3)`



- □ `test` method has access to its local variable (e.g. `x`) as well as the captured variables (e.g. `y` and `A.this`)
- □ Java only allows a local class to capture variables that are explicitly declared **`final`** or effectively (implicitly) final
  - – an effectively final variable is one whose value does not change after initialization

A Note on `Optional`'s `of` and `empty`

- □ `Optional` allows `of` and `empty` can be called anywhere in the pipeline, thereby rendering previous operations obsolete! ☹

```
jshell> Optional.of("abc").map(x -> x.length()).of(1.23)
$.. ==> Optional[1.23]
```

```
jshell> Optional.of("abc").map(x -> x.length()).empty()
$.. ==> Optional.empty
```

- □ Define static method `of` from a `Maybe` interface instead, e.g.

```
jshell> interface Maybe<T> {
   ...>     static <T> Maybe<T> of(T t) {
   ...>         return new Maybe<T>() {};
   ...>     }
   ...> }
|  created interface Maybe
```

```
jshell> Maybe.<Integer>of(1)
$.. ==> Maybe$1@7530d0a
```

```
jshell> Maybe.<Integer>of(1).of("one")
|  Error:
|  illegal static interface method call
|    the receiver expression should be replaced with the type qualifier 'Maybe<java.lang.Integer>'
|  Maybe.<Integer>of(1).of("one")
|  ^---------------------------^
```

The `Maybe` Interface

```
interface Maybe<T> {

    static <T> Maybe<T> of(T value) {
        return new Maybe<T>() { // inner class implementation; can define lambda instead?
            private T get() {
                return value; // value is captured from the enclosing method
            }

            private boolean isEmpty() {
                return this.get() == null;
            }

            // other private methods

            public Maybe<T> filter(Predicate<? super T> predicate) {
                return this.isEmpty() ? this :
                    predicate.test(this.get()) ? this : Maybe.<T>empty();
            }

            // other public methods

            @Override
            public String toString() {
                return this.isEmpty() ? "Maybe.empty" : "Maybe[" + this.get() + "]";
            }
        };
    }

    static <T> Maybe<T> empty() {
        return Maybe.<T>of(null);
    }

    public Maybe<T> filter(Predicate<? super T> predicate);

    // other public method specifications
}
```