CS2030 Lecture 8

Computation Context

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

Semester 2 2022 / 2023

Lecture Outline and Learning Outcomes

- 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
- Awareness of *variable capture* associated with a *local class*
- Understand variable capture using the Java memory model

Defining a *Maybe* Context

```java
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 + "]";
        }
    }
```

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

- To be declared as private helper methods

```
private T get() {
    return value; //this.get();
}

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

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

- Although Java's `Optional` declares these methods with public access, **you should avoid using them**
  - programming with contexts should be *declarative* rather than imperative

Imperative vs Declarative Programming

- *Imperative* programming specifies *how* to do a task

```
boolean circleContainsPoint(Optional<Circle> oc, Point point) {
    if (oc.isEmpty()) {
        return false;
    } else {
        return oc.get().contains(point);
    }
}
```

  - the above requires awareness of a value (or state) in the context, and checking whether there is a value in the context so as to take it out for further processing

- Declarative programming simply specifies *what* to do

```
boolean circleContainsPoint(Optional<Circle> oc, Point point) {
    return oc.map(x -> x.contains(point)).orElse(false);
}
```

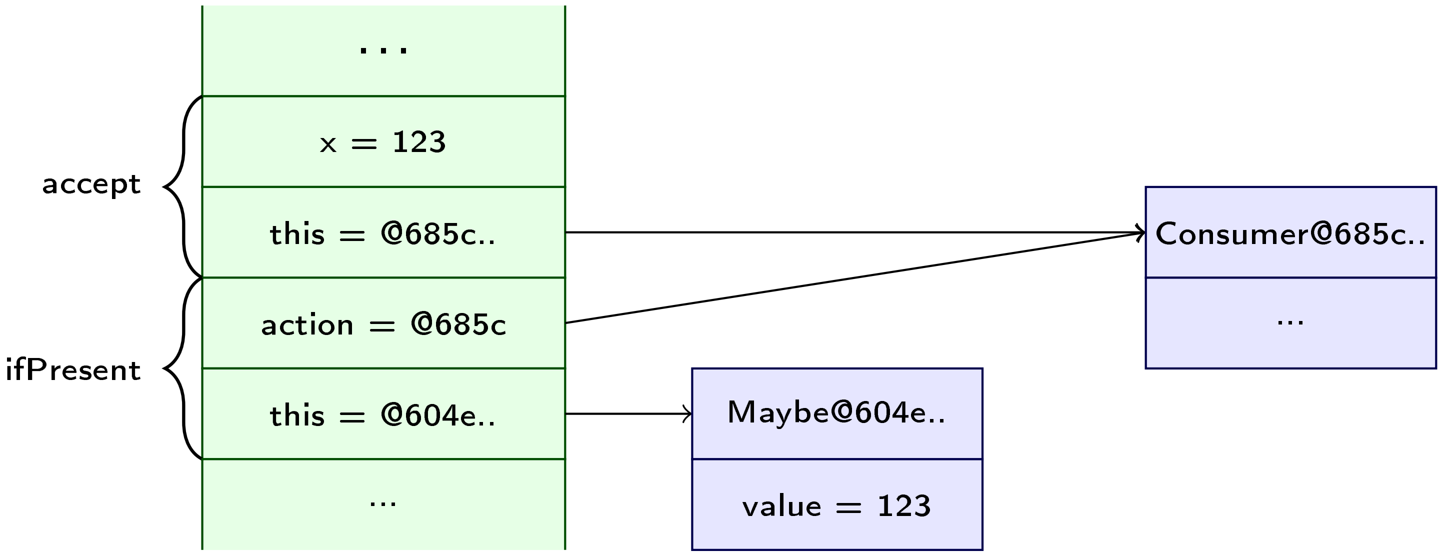
`ifPresent` Method

- Define the following `ifPresent` method in `Maybe` class

```
public void ifPresent(Consumer<? super T> action) {
    if (this.isPresent()) {
        action.accept(value); // snapshot upon calling accept
    }
}
```

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

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



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` and `map` Methods

- Define the `filter` method with nested conditional expressions

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

```
jshell> Maybe.<Integer>empty()
$.. ==> Maybe.empty

jshell> Maybe.<Integer>of(123).filter(x -> x % 2 == 1)
$.. ==> Maybe[123]

jshell> Maybe.<Integer>of(123).filter(x -> x % 2 == 0)
$.. ==> Maybe.empty
```

- Define the `map` method

```java
public <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]
```

Overriding `equals` Method in `Maybe`

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    } else if (obj instanceof Maybe<?> other) {
        if (this.isEmpty()) {
            return other.isEmpty();
        } else {
            return !other.isEmpty() && this.get().equals(other.get());
        }
    } 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

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

- Java's `Optional` allows `of` and `empty` to 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
```

- Call a static method from an interface instead, e.g.

```
jshell> interface Foo<T> {
   ...>     static <T> Foo<T> of() {
   ...>         return new Foo<T>() {}; // use an anonymous inner class!
   ...>     }
   ...> }
|  created interface Foo
```

```
jshell> Foo.<Integer>of()
$.. ==> Foo$1@52cc8049
```

```
jshell> Foo.<Integer>of().of() // of can only be called at the start :)
|  Error:
|  illegal static interface method call
|    the receiver expression should be replaced with the type qualifier 'Foo<java.lang.Integer>'
|  Foo.<Integer>of().of()
|  ^--------------------^
```

The `Maybe` Interface

```java
interface Maybe<T> {

    static <T> Maybe<T> of(T value) {
        return new Maybe<T>() { // inner class implementation of Maybe
            private final T v = value; // setting the property directly

            private T get() {
                return this.v;
            }

            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);
    }

    Maybe<T> filter(Predicate<? super T> predicate);
    // other public method specifications
}
```

Local Class and Variable Capture

- Consider the following slight modification

```
interface Maybe<T> {

    static <T> Maybe<T> of(T value) {
        return new Maybe<T>() {
            private T get() {
                return value; // value is captured!
            }
            ...
```

- The program compiles as Java supports variable capture in local classes
  - an anonymous inner class is a local class — a class that is declared locally within a code block, typically a method block
  - variables declared outside of the local class (in the surrounding block) are captured into the local class

Local Class and Variable Capture

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

```
jshell> class A {
   ...>     private final int x;
   ...>     A(int x) {
   ...>         this.x = x;
   ...>     }
   ...>     Function<Integer,Integer> f(int y) {
   ...>         return new Function<Integer,Integer>() {
   ...>             @Override
   ...>             public Integer apply(Integer z) {
   ...>                 return A.this.x + y + z;
   ...>             }
   ...>         };
   ...>     }
   ...> }
|  modified class A
```

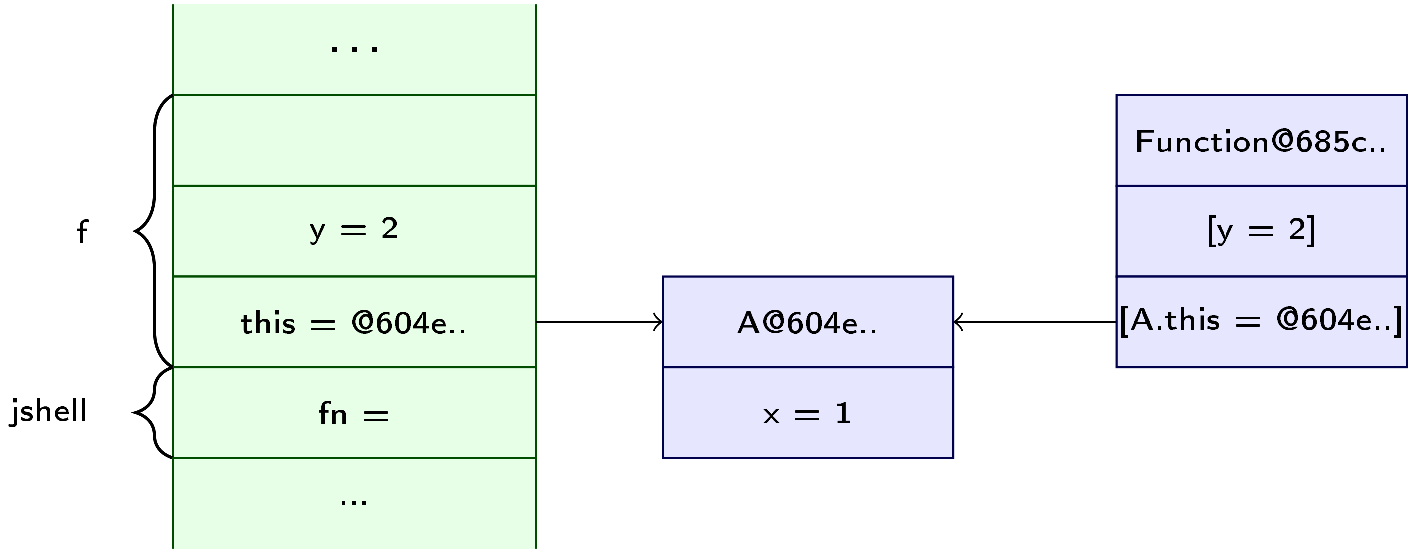
- *Variable capture*: local class makes a copy of variables of the enclosing method and reference to the enclosing class
- `A.this` is known as a *qualified this*

Java Memory Model

- Memory model of the statement

  `jshell> Function<Integer,Integer> fn = new A(1).f(2)`

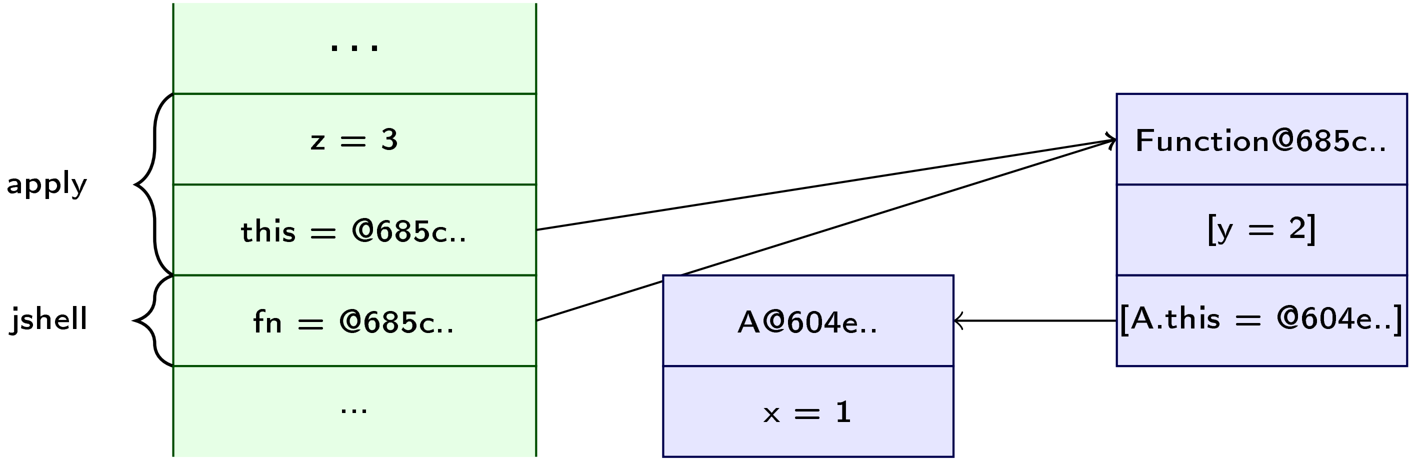
  just before returning from the method `f`



- *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

Java Memory Model

- Memory model upon invoking the method `fn.apply(3)`



- `apply` method has access to its local variable (e.g. `z`) 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

Exercise

- Consider the following class `A`

```
jshell> class A {
   ...>     Integer apply(int x) {
   ...>         return x * 10;
   ...>     }
   ...>     Function<Integer,Integer> f(int y) {
   ...>        return new Function<Integer,Integer>() {
   ...>           @Override
   ...>           public Integer apply(Integer z) {
   ...>              return A.this.apply(z) + y;
   ...>           }
   ...>        };
   ...>     }
   ...> }
|  modified class A
```

- What is the outcome of `new A().f(2).apply(3)`?
- Now replace `A.this.apply(z)` in method `foo` with `this.apply(z)`. Does it compile?
  - what is the outcome of `new A().f(2).apply(3)` now?