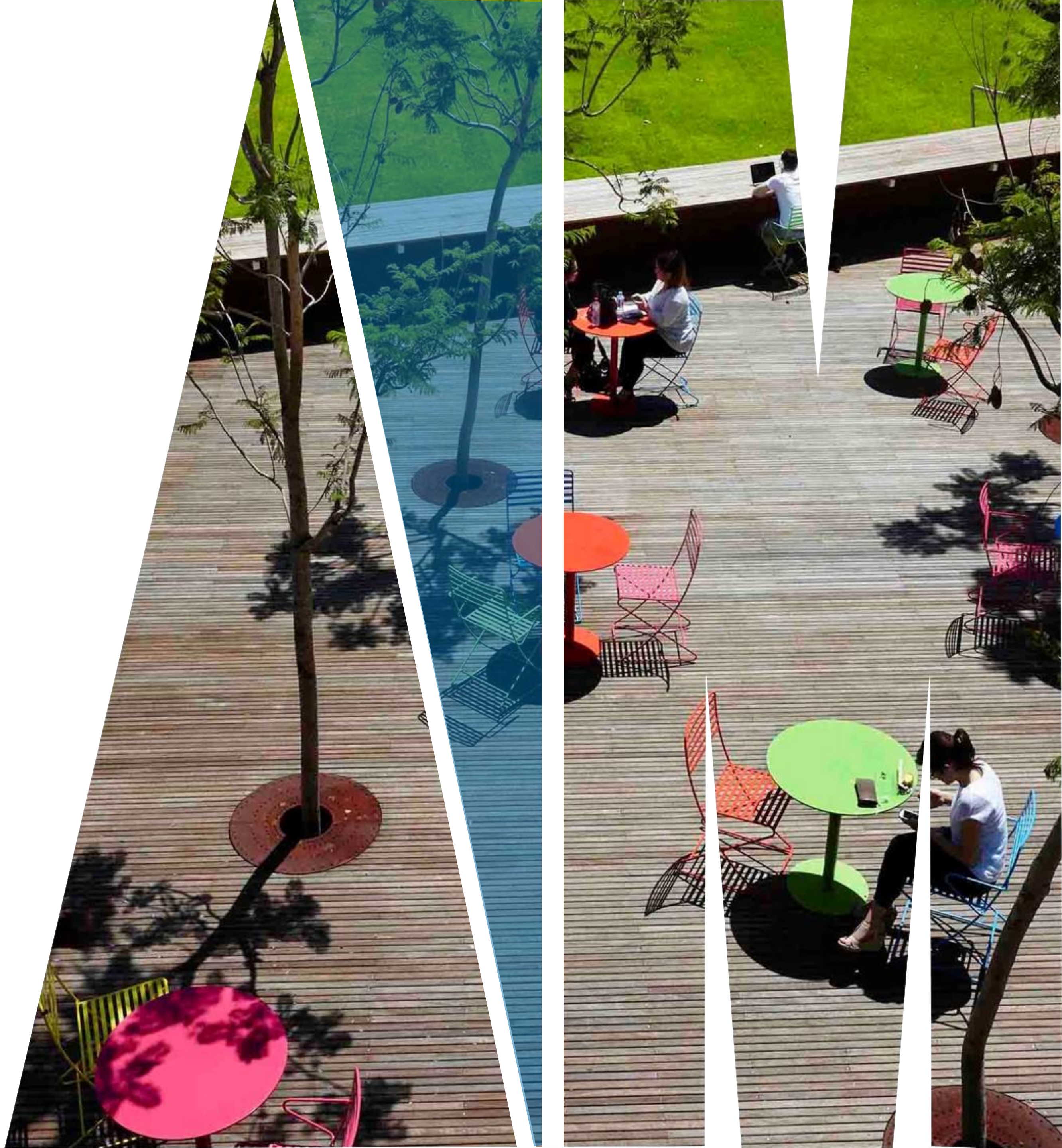
# FIT2099 Object-Oriented Design and Implementation

## Interfaces



## Outline

Interfaces

- Real world examples

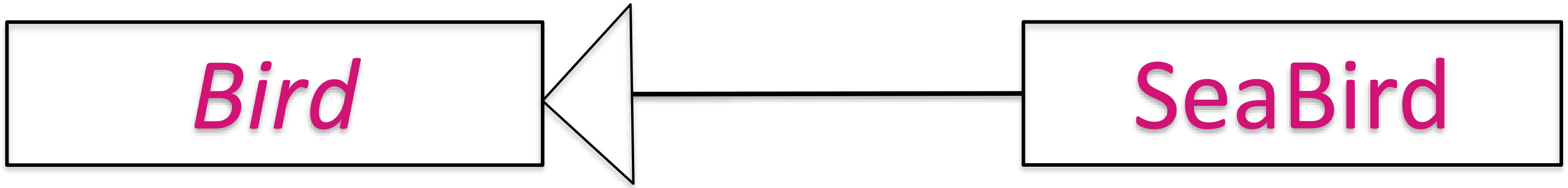
Interfaces versus - concrete and abstract classes

Default methods

UML representation

INTERFACES

# REAL-WORLD REPRESENTATION



**Seagull**

**Can swim and fly**

**Frigatebird**

**Can fly** but cannot swim, they drawn if they fall into the water

**Penguin**

**Can swim** but, definitively, cannot fly.

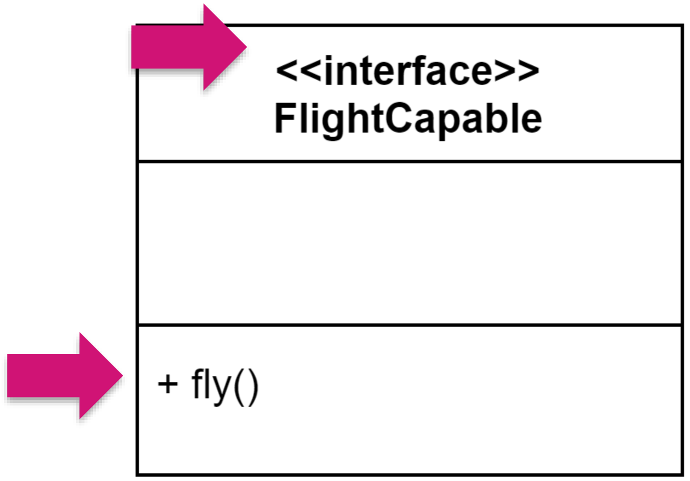
# THE INTERFACE

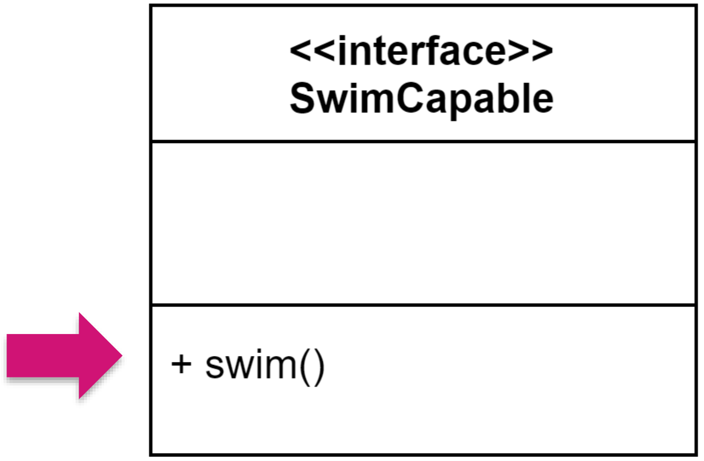
An interface is a **completely "abstract class"** that is used to group related methods with **empty bodies.**

In other words, they are **collections of abstract methods.**

Like a pure **blueprint** of what a class must do and not how.

# INTERFACES
## UML REPRESENTATION





swim()
fly()

fly()

swim()

## CONCRETE/ABSTRACT V/S INTERFACES

| | Interfaces | Abstract classes | Concrete classes |
|---|---|---|---|
| Constructor | ✗ | ✓ | ✓ |
| Static/final attributes | ✓ | ✓ | ✓ |
| Non-static or non-final attributes | ✗ | ✓ | ✓ |
| PRIVATE attributes and methods | ✗ | ✓ | ✓ |
| PROTECTED attributes and methods | ✗ | ✓ | ✓ |
| PUBLIC methods | ✓ | ✓ | ✓ |
| ABSTRACT methods | ✓ | ✓ | ✗ |
| STATIC methods | ✓ | ✓ | ✓ |
| FINAL methods | ✗ | ✓ | ✓ |
| DEFAULT methods | ✓ | ✗ | ✗ |
| Multiple inheritance? | ✓ | ✗ | ✗ |

## INTERFACES
## SYNTAX

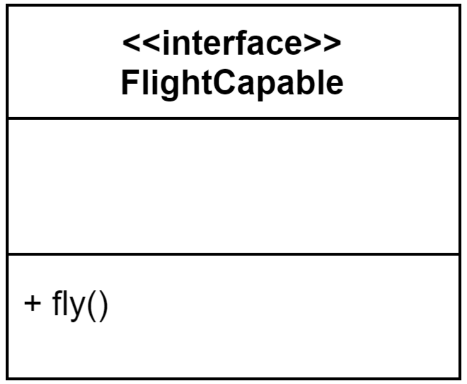
```
interface interface_name {

    // declare final/static attributes
    // declare methods that are abstract
    // by default.
}
```

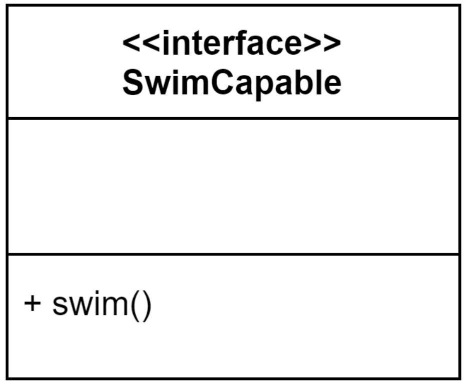
# INTERFACES
# SYNTAX

Interface methods do not have a body.

```
interface FlightCapable{
  public void fly();
}
```

```
interface SwimCapable{
  public void swim();
}
```





INTERFACES

# SYNTAX

Penguin

swim()

The interface must be "**implemented**" (kind of like inherited) by another class with the **`implements`** keyword (instead of extends).

```
interface SwimCapable{
  public void swim();
}


class Penguin extends SeaBird implements SwimCapable{
  public void swim() {
    // The body of swim() is provided here
    System.out.println("The penguin can swim");
  }
}
```

The body of the interface method is provided by the "implement" class.

# EXTENDING MULTIPLE INTERFACES

The extends keyword is used once, and the parent interfaces are declared in a **comma-separated list**.

```
interface FlightCapable{
  public void fly();
}
```

```
interface SwimCapable{
  public void swim();
}
```

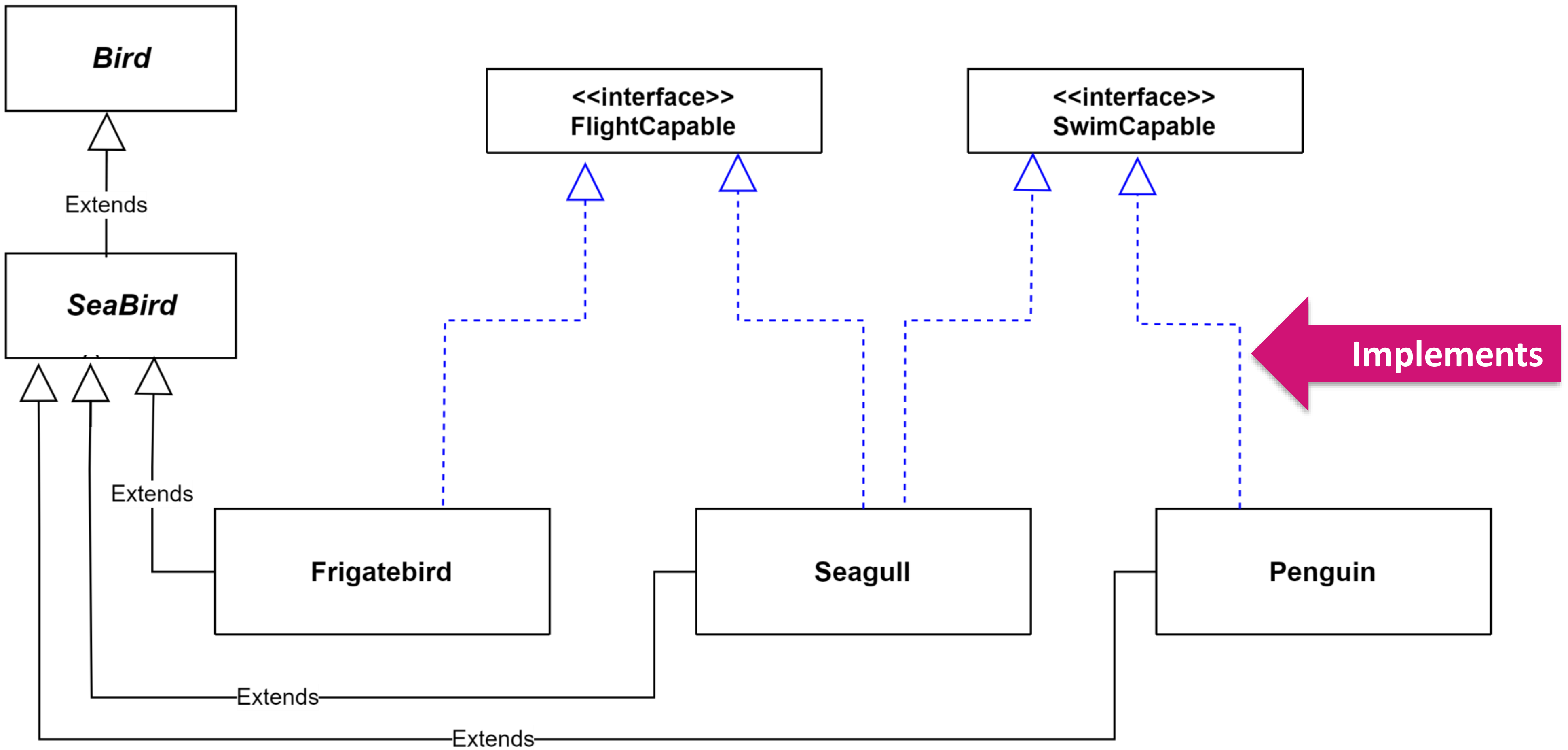
```
class Seagull extends SeaBird implements SwimCapable, FlightCapable{
    public voic swim() {
        System.out.println("The Seagull can swim");
    }

    public voic fly() {
        System.out.println("The Seagull can fly");
    }
}
```

**Seagull**

**swim()**
**fly()**

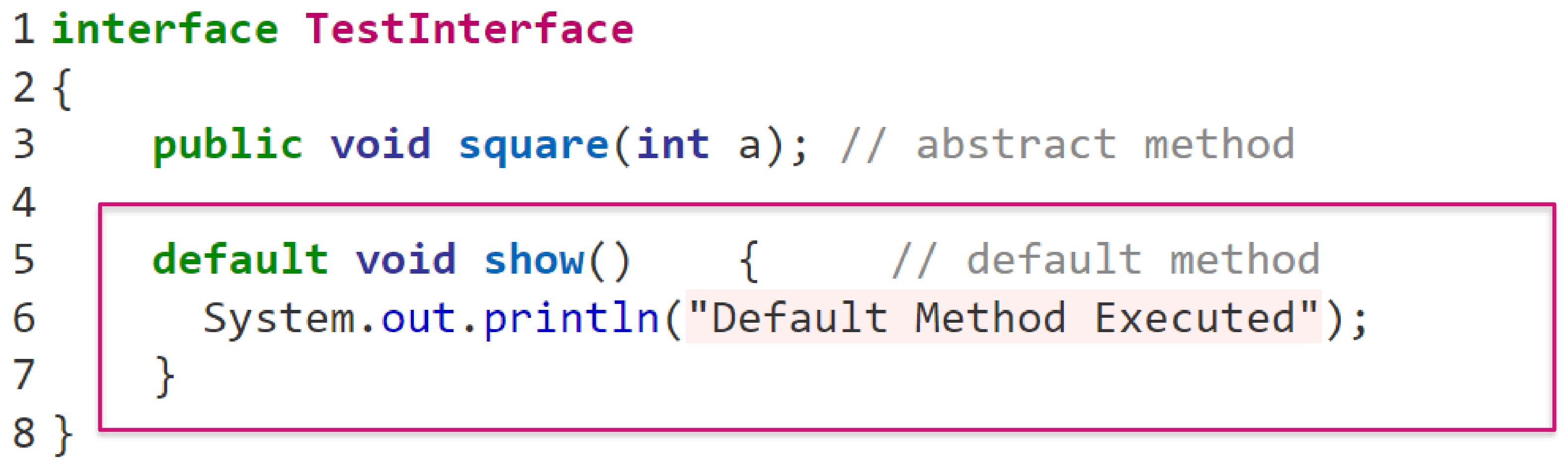
INTERFACES

## UML REPRESENTATION



## DEFAULT METHOD IN INTERFACES

In JAVA 8 and above, **default methods** allow the interfaces to have methods with implementation without affecting the classes that implement the interface.



```java
interface TestInterface
{
    public void square(int a); // abstract method

    default void show()    {      // default method
      System.out.println("Default Method Executed");
    }
}
```

The most common use of interface default methods is to **incrementally provide additional functionality to a given type without breaking down the implementing classes.**

Default methods are also known as **defender methods** or **virtual extension methods**.

## REASONS FOR USING INTERFACES

Interfaces are used to achieve **abstraction**.

Designed to support **dynamic method resolution at run time**

It helps you to achieve **loose coupling**.

Allows you to **separate** the definition of a method from the inheritance hierarchy

## Summary

Interfaces

  Real world examples

Interfaces versus - concrete and abstract classes

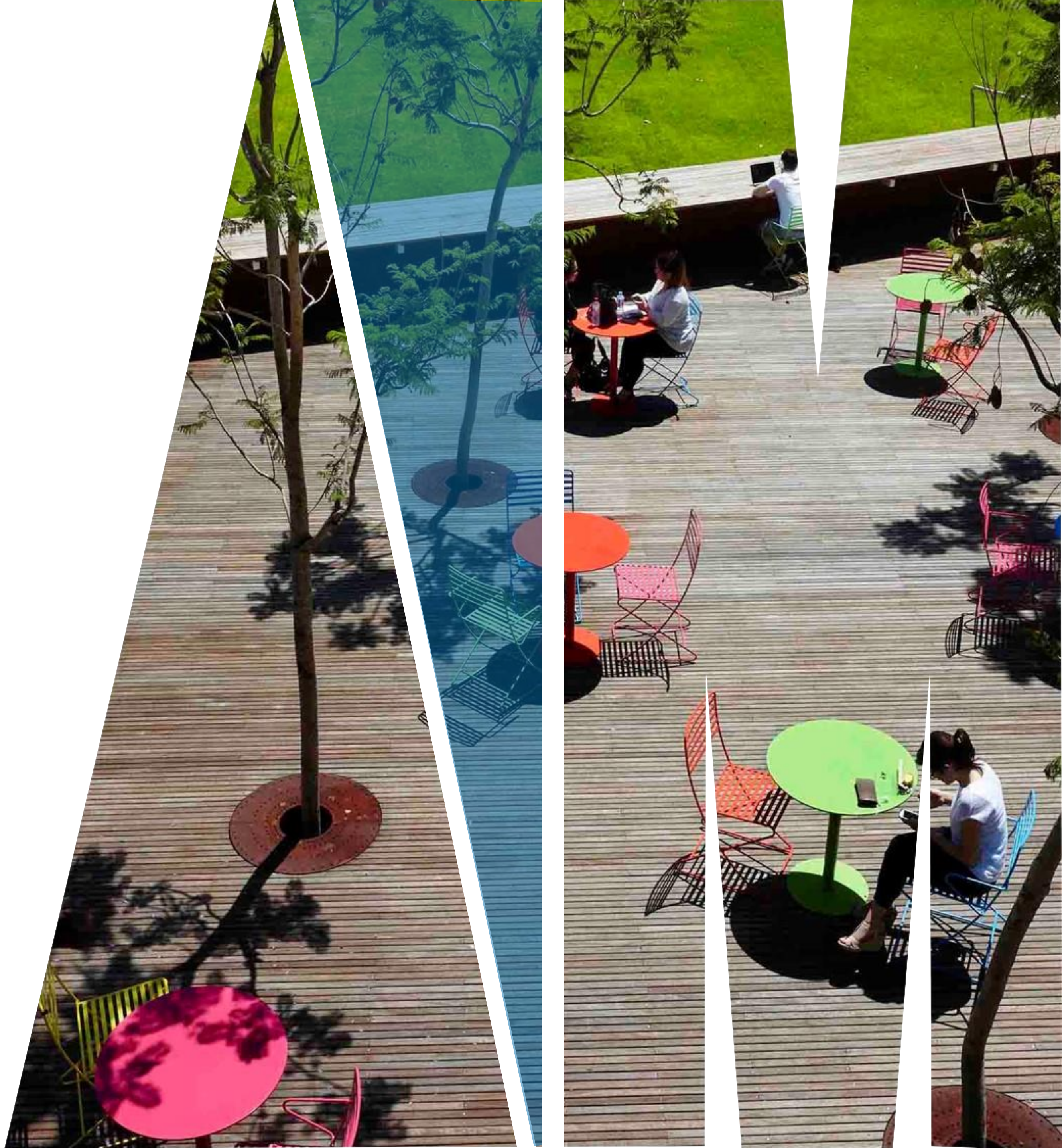
Default methods

UML representation

MONASH University

MONASH
University

Thanks



MONASH
University