Applied Programming I

Programming computers in a nutshell.

Introduction to Object-Oriented Programming (OOP)

Basics only

Today's objectives

- Acquiring the basic idea of programming paradigms
- Understand principles of object-oriented programming

Programming paradigms

Imperative vs. declarative programming

- In **imperative programming**, the program code controls the execution flow and the program's state by commands (Latin *imperare*, to command).
- Whereas in **declarative programming**, the results that the program should achieve are described. (Latin: *declarare*, to declare).

Procedural and structural programming

We have used these up to now.

- Both are subcategories of imperative programming.
- **Procedural programming** focusses on the use of procedures (other term for functions).
- **Structural programming** focusses on the flow of control using selection and iteration.
- Most modern languages comprise both paradigms (and more).

Is Matlab procedural and structural?

What do you think?

Matlab: function, iteration and selection

```
function max = findMax(x) % x is array
  max = x(1);
  for i = 2:size(x,2)
    if max < x(i)
      max = i;
    end
  end
end
```

Object-oriented programming (OOP)

- **Object-oriented programm**ing is a subset of imperative programming. It incorporates structured and procedural programming features but adds the concept of objects.
- **Objects** are abstract data types that have fields (like structs) and methods (functions). Classes describe how objects is designed.
- Inheritance, polymorphism, abstraction, encapsulation are features of OOP.
- Many languages are only understandable by their usage of OOP-concepts, such as C# or Java, to some extend Python, too.
- OOP is covered later.

Examples

- C does not support object-oriented programming by default. Actually, C++ started as an extension of C in order to support this.
- JavaScript supports object-oriented programming.
- Python takes the object-oriented paradigm serious, everything is an object.
- C# is designed for object-oriented programming.

Object-oriented programming (OOP)

Object-oriented programming is a subset of imperative programming. It incorporates structured and procedural programming features but adds these concepts by ‘objects’ :

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

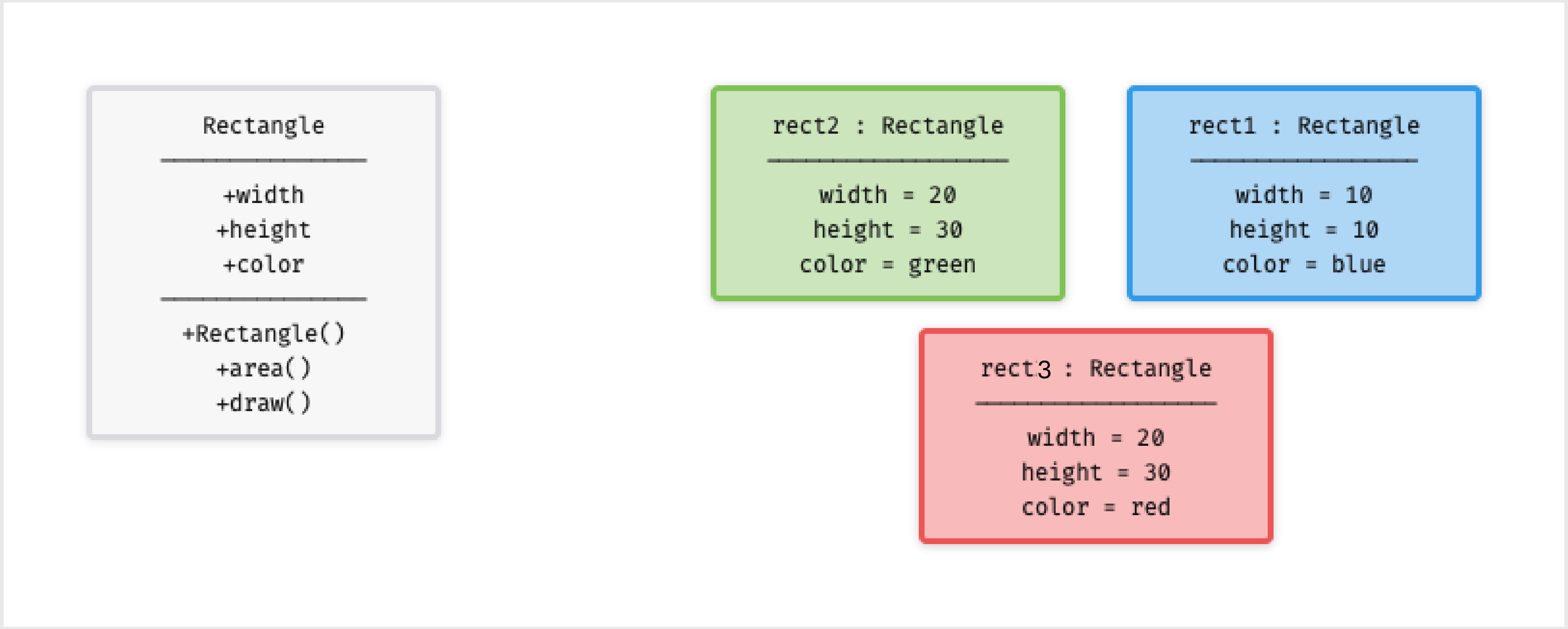
Abstraction

- **Abstraction** offers a way to view on an entity, an object, while focusing on its essential features and ignoring its details.
- Depending on the viewpoint, abstraction can have different layers.
- The concept **class** and **object** are used as an abstraction mechanism in OOP in order to model an entity.

Class and object

- A **class** is a kind of user-defined datatype that has a structure given by variables (called **attributes**) and a behavior given by functions (called **methods**).
- A class abstracts an entity / an object. It defines an essential **interface** and hides complex implementation (information hiding).
- An **object** is a variable (called instance) of type class.
- Each object has its own state given by the values of its attributes.

Example class and objects of this class



Example: Simple class in C#

```
// an object/instance of the class
Rectangle rect = new Rectangle();
// access the attributes
rect.Width = 10;
rect.Height = 10;
// call methods
rect.Draw();
Console.WriteLine(rect.Area());
Console.WriteLine(rect.ToString());

// this is a class
public class Rectangle
{
  // attributes
  public int Width, Height;
```

```
// methods
  public float Area() {
    return (float)Width * (float)Height;
  }

  public void Draw() {
    for (int i = 0; i < Height; i++) {
      for (int j = 0; j < Width; j++) {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }
}
```

Note: C# usually has an OOP-kind entry

```
class Program
{
  // Main method - entry point of the application
  static void Main(string[] args) {
    // an object/instance of the class
    Rectangle rect = new Rectangle();
    // access the attributes
    rect.Width = 10;
    rect.Height = 10;
    // call methods
    rect.Draw();
    Console.WriteLine(rect.Area());
    Console.WriteLine(rect.ToString());
  }
}
```

Constructors and destructors

- A **constructor** in oop is a special method that is called when an instance of a class is created.
- A constructor has the purpose to initialize the object.
- A **destructor** is the method that is called when an instance of a class is destroyed.
- A destructor has the purpose to release memory.
- Destructors are important in languages that leave the memory handling to the programmer.

Encapsulation

- **Encapsulation** is related to abstraction. It not only let you ignore details but does even not allow to get to some of the details.
- A capsule is build around your entity. Information is only available at a level that the programmer defines. Information hiding is taken a step further.
- Access control is designed and implemented by the programmer.
- Access can be **public** (everyone), **protected** (only in inherited classes – upcoming), or **private** (only in the class itself).

Example: C# constructor and encapsulation

```
// an object/instance of the class
Rectangle rect = new Rectangle(10, 10);
// access the attributes would
// be an error
// int w = rect.Width;
// call methods
rect.Draw();
Console.WriteLine(rect.Area());
Console.WriteLine(rect.ToString());

// this is a class
public class Rectangle
{
 // attributes, cannot be accessed from outside
 // because they are private
 private int Width, Height;

 // constructor
 public Rectangle(int w, int h) {
   Width = w;
   Height = h;
 }

 // methods
 public float Area() {
   return (float)Width * (float)Height;
}
```

Self-reference in objects

- Objects often need to work with their own attributes. Thus, programming languages provide keywords as a reference of the instance to itself.
- JavaScript, C++ and C# use the keyword `this`.
- Python uses the keyword `self`.

Example: Python class

```
# Python needs self to access instance attributes / methods
class Rectangle:
  # Constructor/initializer: __...__ marks special methods
  def __init__(self, width, height):
    # but __ prefix marks private
    self.__width = width
    self.__height = height

  # __ marks private
  def __i_am_private(self) :
    print("private")

  # Method to calculate area
  def area(self):
    return self.__width * self.__height

  # Method to draw the rectangle
  def draw(self):
    self.__i_am_private()
    for i in range(self.__height):
      print('*' * self.__width)

  # String representation of the object
  def __str__(self):
      return f"R(width={self.__width},height={self.__height})"

# Create an instance of the class with width and height
rect = Rectangle(10, 10)
# Call methods
rect.draw()
print(rect.area())
print(rect)
#rect.__i_am_private() does not work
```

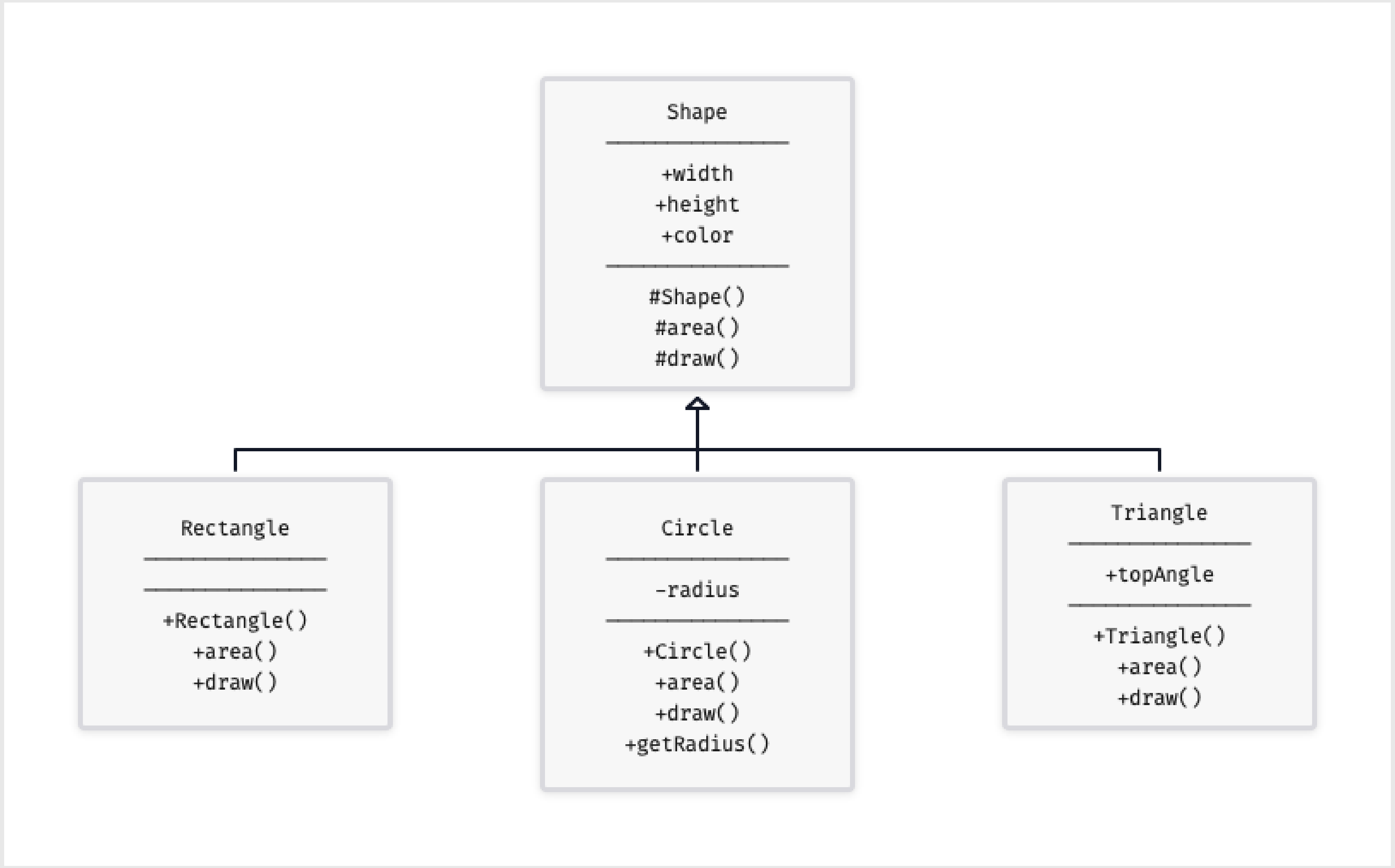
Example: JavaScript class

```
class Rectangle {
 // Private fields
 #width; #height;

 // Constructor
 constructor(width, height) {
  this.#width = width;
  this.#height = height; }

 // Private method
 #iAmPrivate() {
  console.log("This is a private method"); }

 // Public method to calculate area
 area() {
  return this.#width * this.#height; }

 // Public method to draw the rectangle
 draw() {
  for (let i = 0; i < this.#height; i++) {
   console.log('*'.repeat(this.#width));
  } }

 // Public method to get string representation of the object
 toString() {
  return `R(width=${this.#width}, h. eight=${this.#height})`;
  }
}
// Create an instance of the class with width and height
const rect = new Rectangle(10, 10);
// Call methods
rect.draw();
console.log(rect.area());
console.log(rect.toString());
```

Inheritance

- Inheritance supports abstraction. It allows a class to **inherit** attributes and methods from another class.
- Building a **hierarchy**, a derived class (child class) inherits from a base class (parent class).
- The derived class contains all information of the base class.
- Similarities go in the base class and differences in the derived classes.
- Base classes can be **abstract**. That means they cannot be instantiated and work as an interface.

Abstract base class with three derived classes



Example: C++ class

```
// base class
class Shape {
  public:
    unsigned int width = 0;
    unsigned int height = 0;
    Color color = Color::WHITE;
    // class is pure virtual / abstract,
    // cannot be instantiated
    virtual float area() = 0;
    virtual void draw() = 0;  protected:
    Shape(int w, int h) : width(w), height(h) { }
};
```

```
// derived class, can be instantiated
class Rectangle : public Shape {
  public:
    Rectangle(int w, int h) : Shape (w, h) {}
    float area() { return width * height; }
    void draw() { /* .. */ }
};
int main() {
  // Shape* s = new Shape(); // not allowed, abstract
  Rectangle* rect = new Rectangle(10, 10);
  std::cout << " " << rect->area() << std::endl;
  delete rect;
  return 0;
}
```

Polymorphism (overriding)

- Polymorphism supports abstraction, too.
- Polymorphism allows to use the interface of a common base class of objects of its derived classes without knowing the type of the derived class.
- The object can have many forms, it can be **polymorphic**.
- A single **interface** (of the base class) can be differently implemented (by the derived classes). This is called **overriding**.

Example: polymorphism in C++

```
// base class
class Shape {
public:
  unsigned int width = 0;
  unsigned int height = 0;
  virtual float area() = 0;
protected:
  Shape(int w, int h) : width(w), height(h) { }
};

// derived classes
class Rectangle : public Shape {
public:
  Rectangle(int w, int h) : Shape (w, h) {}
  float area() { return width * height; }
};
```

```
class Circle : public Shape {
public:
  Circle(int w, int h) : Shape (w, h) {}
  float area() {return float(height)/2*float(height)/2*M_PI; }
};

int main() {
  Rectangle* rect = new Rectangle(10, 10);
  Circle* circle = new Circle(15, 15);
  Shape* shape = circle;
  std::cout << " " << shape->area() << std::endl; // 176.715
  shape = rect;
  std::cout << " " << shape->area() << std::endl; // 100
  delete rect;
  delete circle;
  return 0;
}
```

Example: polymorphism in C#

```
using System;
// base class
abstract class Shape {
  public uint Width { get; set; }
  public uint Height { get; set; }
  protected Shape(int width, int height) {
  Width = (uint)width;  Height = (uint)height; }
  public abstract float Area();
}

// derived classes
class Rectangle : Shape {
  public Rectangle(int width, int height) : base(width, height) { }
  public override float Area() { return Width * Height; }
}

class Circle : Shape {
  public Circle(int width, int height) : base(width, height) { }
  public override float Area() {
    return (float)(Height / 2.0 * Height / 2.0 * Math.PI); }
}

class Program {
  static void Main() {
    Rectangle rect = new Rectangle(10, 10);
    Circle circle = new Circle(15, 15);
    Shape shape = circle;
    Console.WriteLine(" " + shape.Area()); // 176.715
    shape = rect;
    Console.WriteLine(" " + shape.Area()); // 100
  }
}
```

Example: polymorphism in JavaScript

```
// base class
class Shape {
  constructor(w, h) {
    this.width = w;
    this.height = h; }
  // This should be overridden by derived classes
  area() {
    throw new Error('should be overridden'); }
}

// derived classes
class Rectangle extends Shape {
  constructor(w, h) { super(w, h); }
  area() { return this.width * this.height; }
}
```

```
class Circle extends Shape {
  constructor(w, h) { super(w, h); }
  area() { return (this.height / 2) *
         (this.height / 2) * Math.PI; }
}

// usage
const rect = new Rectangle(10, 10);
const circle = new Circle(15, 15);
let shape = circle;
console.log(shape.area()); // Output: 176.7145....
shape = rect;
console.log(shape.area()); // Output: 100
```

Example: polymorphism in Python

```
from math import pi

# base class
class Shape:
  def __init__(self, w, h):
    self.width = w
    self.height = h

  def area(self):
    raise NotImplementedError("Subclasses should do this!")

# derived classes
class Rectangle(Shape):
  def __init__(self, w, h):
    super().__init__(w, h)

  def area(self):
    return self.width * self.height
```

```
class Circle(Shape):
  def __init__(self, w, h):
    super().__init__(w, h)

  def area(self):
    return (self.height / 2) ** 2 * pi


# main functionality
if __name__ == "__main__":
  rect = Rectangle(10, 10)
  circle = Circle(15, 15)
  shape = circle
  print(f" {shape.area()}")
  shape = rect
  print(f" {shape.area()}")
```

Class variables vs instance variables

- **Class variables** (or static variables) are available for all instances of a class. They are not instance specific. They can also be accessed without class instance at all.
- **Instance variables** are specific for each instance of a class. An instance can only access it own instance variables.
- Corresponding to variables, there are also **class methods** and **instance methods**.

Example: class variable in JavaScript

```
class Rectangle {
 // class variable
 static numberOfRectangles = 0;

 // Private fields
 #width;
 #height;
 // Constructor
 constructor(width, height) {
 this.#width = width;
 this.#height = height;
 Rectangle.numberOfRectangles++;
}
```

```
// Create an instances of the class
const rect1 = new Rectangle(10, 10);
console.log(Rectangle.numberOfRectangles); // 1
const rect2 = new Rectangle(10, 20);
console.log(Rectangle.numberOfRectangles); // 2
```

OOP-Concepts in brief according to Microsoft

- *Abstraction* Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.
- *Encapsulation* Hiding the internal state and functionality of an object and only allowing access through a public set of functions.
- *Inheritance* Ability to create new abstractions based on existing abstractions.
- *Polymorphism* Ability to implement inherited properties or methods in different ways across multiple abstractions.

[Source: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop, accessed 24-08-29]

Your task: develop a simple simulation Description

The simulation is about a service desk that has (for now) one agent who serves customers.

The customer arrive at different times at the desk and have issue that are defined by the time they will take the agent. Thus, the agent can be busy or not.

If the agent is busy, the customers queue up. The agent processes the queue step by step.

Your task: develop a simple simulation Example

Time t	Events at begin of t	Customer queue after t	Agent
0	-	Empty	Free
1	Customer C1 with issue of duration 3 arrives Agent starts serving C1	C1	Busy with C1
4	Agent finishes serving C1 Customer C2 with issue of duration 3 arrives Agent starts serving C2	C2	Busy with C1
5	Customer C3 with issue of duration 4 arrives	C2, C3	Busy with C2
7	Agent finishes serving C2 Agent starts serving C3	C3	
11	Agent finishes serving C3	Empty	Free

Your task: develop a simulation Step I

- In your project groups, develop a program step by step that has
  - Two different kinds of persons: agent and customer. Both have in common that they have ids.
  - Two different kinds of events: arrival and service. The events share that they are generated with a time, contain information about the customer. Besides, the service has a expiration time.
- Design the classes first. Then program them, and write code to test them. The languages to be used will be assigned to you.
- Time: 45 minutes.

Break

10 minutes

One possible approach… JavaScript

```
class Person {
  constructor(id) {
  this.id = id;
  }
}

class Agent extends Person {
  constructor(id) {
  super(id);
  this.busy = false;
  }
}

class Customer extends Person {
constructor(id, duration) {
  super(id);
  this.duration = duration;
  }
}
```

And some other part … in Python

```
class Event:
  def __init__(self, time, customer):
    self.time = time
    self.customer = customer


class ArrivalEvent(Event):
  def __init__(self, time, customer):
    super().__init__(time, customer)
```

```
class ServiceEvent(Event):
  def __init__(self, time, customer):
    super().__init__(time, customer)

  def end(self):
    return self.time + self.customer.duration
```

Your task: develop a simulation
Step 2

- Now write the program logic.
- You might need at least a queue and a loop processing the events or/and time steps.
- Perhaps you need rework your class design.
- If you are done with the program, use Copilot or another LLM tool, to translate your into another programming language.

- Time: 75 minutes.

Possible logic part in JavaScript

```
// ... time based, loop over time
for (let t = 0; t <= maxTime; t = t+1) {
 let nextEvents = [];              // queue for later
 let currentEvents = [];           // queue for this time
 for (let event of pendingEvents) { //event in queue?
   if (event.isDue(t)) {          // and due?
     currentEvents.push(event);    // then take now
   } else {
     nextEvents.push(event);      // else later
   }
 }
 let newEvents = []; // queue for new created events
 for (let event of currentEvents) {
   let newEvent = event.process(t, queue); // process
   if (newEvent != null) {        // null return, done
     newEvents.push(newEvent);     // else save for later
   }
 }
 for (let event of newEvents) { // any created events
   if (event.isDue(t)) {       // immediately due?
     let nextEvent = event.process(t, queue);
     if (nextEvent) {         // new created for later
       nextEvents.push(nextEvent);
     }
   }
   else {
     nextEvents.push(event);
   }
 }
 pendingEvents = nextEvents; // copy queue of later event
}                              // for next time step
```

Questions ?

Generated with AI