COMP10062: Week 3 Guide

Sam Scott, Mohawk College, 2020

0.1 Reading for this Week

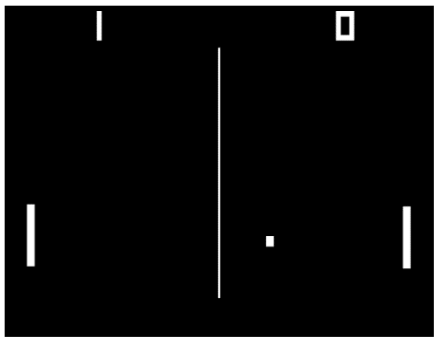
For this week, you should **read sections 5.1 to 5.3 and Appendix 5**.

0.2 What is Object Oriented Programming?

In your Python course, you engaged in procedural (aka "structured") programming. In procedural programming, you view a programming problem as a series of tasks. You break down the problem into subtasks (procedures or functions) and then perform one subtask at a time. Below is an example of how to view the classic game of Pong in a procedural way. Steps 1 through 7 could each be a function call.

Example: A simple Pong game

1. clear the screen
2. update ball position
3. update paddle positions (if user input)
4. draw the screen
5. detect ball collision (paddles, walls)
6. change ball direction (if necessary)
7. update score (if necessary)
8. go to step 1



In Java, you will usually engage in object-oriented programming. In object-oriented programming, you view a programming problem in terms of the objects involved. You analyze the problem, decide what different types of objects are involved, decide on what **attributes** each type has and decide what **behaviours** each type exhibits. Then you write a blueprint called a **class** for each object type, create the objects (**instances** of the class) from the main program and in some cases (like the Pong example) you may even be able to exit the main method and let the objects interact with one another on their own.

The Pong example

1. create a ball object
2. create paddle objects
3. create a scoreboard object
4. create keyboard listener objects
5. create a screen updater object
6. relax and let the objects talk to each other

Note that these Pong examples are highly over-simplified.

What is an Object?

Real-world object: An entity with attributes and behaviours (e.g. a car). An **attribute** is a property of the object (e.g. color, transmission type, amount of gas in tank, etc.). A **behaviour** is something the object does or that you can do with the object (e.g. start engine, accelerate, brake, etc.)

Java Object: A package of **instance variables** or **fields** (*attributes*) and **instance methods** (*behaviours*). For example, a `String` object contains a list of characters (stored in an instance variable) and instance methods such as `length()`, `equals()` and `indexOf()`.

Java Class: A blueprint for creating a particular kind of object. Every Java Class you create becomes a type that you can use to declare variables.

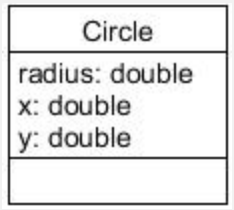
1. Objects and Instance Variables (Sections 5.1 & 5.3)

a. Creating a Class

Java

```
public class Circle {
    double radius = 10.0;
    double x = 100.0;
    double y = 100.0;
}
```

UML Class Diagram



Notes

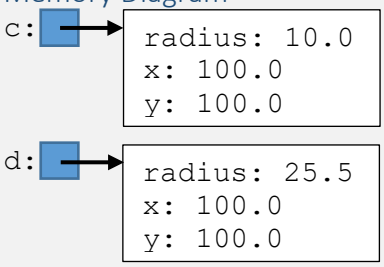
UML Class diagrams show the class name, the names and types of the instance variables, and the names and types of any instance methods. This class has no instance methods.

b. Creating Objects

Java

```
Circle c = new Circle();


Circle d = new Circle();
d.radius = 25.5;
```

Memory Diagram



Notes

Variables `c` and `d` contain references (memory addresses) that allow access to the objects stored in RAM. The "." operator "dereferences" the variable to get to the object.

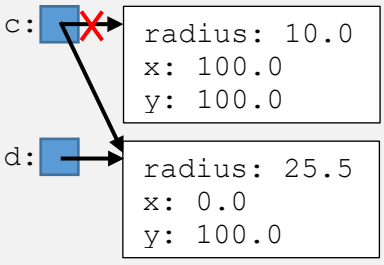
`System.out.println(c)` will show the memory address.

c. Assigning and Comparing Objects

Java

```
System.out.println(c==d)

c = d;
c.x = 0.0;
System.out.println(c.x);
System.out.println(d.x);




System.out.println(c==d)
```

Memory Diagram



Questions

Can you walk through the changes to the diagram in part b given the code in part c?

What will the output of this program be?

Note

Objects with no active reference are automatically disposed of by Java's "Garbage Collection" mechanism.

2. Instance Methods (Sections 5.1 & 5.3)

In Python

```
def addup(a,b,c):
    d = a + b + c
    return d


print(addup(1,2,3))
```

In Java

```
int addup(int a, int b, int c) {
    int d = a + b + c;
    return d;
}

System.out.println(addup(1,2,3));
```

Notes

- A **method** is a function that is declared inside a class definition.
- In Java, all statements (except variable declarations) must be inside a method, and all methods must be inside a class.
- When you run a Java class, the `main` method executes.
- The declaration above starts with the type of the method.
    - This is the type of value that the method must return
    - If you try to return the wrong type, you'll get a syntax error
    - The type can also be `void`, which means the method does not return anything
- Parameters and local variables must be declared using types
- When you call a method, arguments are matched to parameters by position
    - No keyword or optional parameters like Python

In Procedural Programming

Functions perform a specific subtask.

- The subtask is related to the arguments.

Arguments provide all information necessary to perform the subtask.

Return values communicate the result of the subtask.

In Object Oriented Programming

Methods implement a specific **behaviour** of an object.

- The behaviour is related to the **instance variables**.

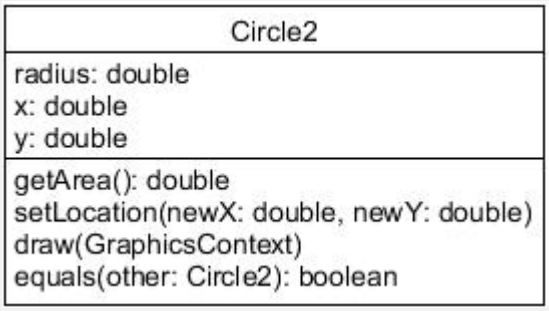
Arguments provide new information not already in the object's instance variables.

Return values send back information from the object. If a behaviour changes an instance variable, we might not need a return value at all.

Java

```
public class Circle2 {

  double radius = 10.0;
  double x = 100.0;
  double y = 100.0;




  double getArea() {
    return Math.PI * radius * radius;
  }




  void setLocation(double newX,
                   double newY) {
    x = newX;
    y = newY;
  }

  void draw(GraphicsContext gc) {
    gc.setStroke(Color.BLACK);
    gc.setLineWidth(radius / 4);
    gc.strokeOval(x - radius,
      y - radius, radius * 2,
      radius * 2);
  }

  boolean equals(Circle2 other) {
    return (x == other.x &&
            y == other.y &&
            radius == other.radius);
    }
}
```

Diagram & Notes



`getArea` has no parameters because all the information is already in the object. Returns the result of its computation.

References to "radius" are references to the instance variable defined above.

`setLocation` has parameters for the information that represents the new location.

`draw` requires a parameter so that it knows where to draw.

`equals` works just like the String method. Use it to determine if two different `Circle` objects represent the same circle.

QUESTION: What would happen if this method contained the statement `other.radius = 100`? Draw a diagram of the situation.

Note

The textbook (e.g. pages 272, 328, 342) shows a style of UML class diagram in which the parameter types appear before the variable names instead of after them. This is not standard, so in this course we will stick to the more standard UML style of `paramName: paramType` as shown above.

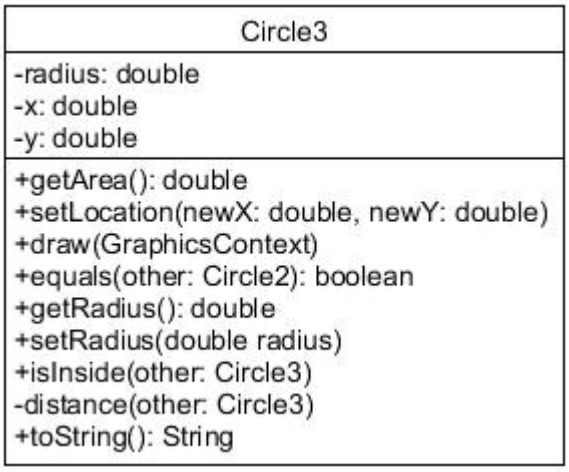
3. Encapsulation (Read Section 5.2)

It's not always a good idea to let other programmers access your instance variables directly:

1. They might change a variable to an illegal value
2. A change to a variable might have other effects within the class.

We avoid this by encapsulating our variables.

To **encapsulate** means to make some instance variables and methods *inaccessible* from outside a class. Usually we make all instance variables **private**, and we make most instance methods **public**.



Accessor methods (get methods) are a standard way to access a private instance variable.

Mutator methods (set methods) are a standard way to set the value of a private instance variable.

In UML class diagrams, "–" means private and "+" means public, as shown above.

See **Circle3.java** on Canvas for the code that matches this diagram.

- Notice that `setRadius()` and `setLocation()` check the values being passed to them and issue warnings or errors. We wouldn't be able to do that if `radius`, `x` and `y` were public.
- Notice that `x` and `y` are "write only". You can change them with `setLocation` but you can't access their values. We also wouldn't be able to do that if `x` and `y` were public.

The toString Method

Circle3.java contains a special `toString()` method. This method returns a `String` representation of an object for debugging purposes. Usually, the `String` contains the class name, plus the values of all the instance variables, like this:

```
Circle3: radius 50.0, location (200.0, 150.0)
```

`toString()` is a kind of "magic" method. It will automatically be called whenever you print or concatenate your variable. So you never have to actually call `toString()`. The system does it for you.

```
System.out.println(circle);          ← will print circle.toString()

String s = "my circle: "+circle;  ← will concatenate circle.toString()
```

(Of course, there's nothing really magic about `toString()`. We'll explain in a later class how this method is really getting called.)

IntelliJ Tip: ALT-INSERT will bring up a menu that lets you create get, set, and toString methods automatically. They will be very basic, so you may have to tweak them to make them do what you want.

The Rectangle Example

Adapted from Listing 5.9 of the textbook…

Java

```
public class Rectangle {
    private int width;
    private int height;
    private int area;

    public void setDimension(int width,
                             int height) {
        this.width = width;
        this.height = height;
        area = width * height;
    }

    public int getArea() {
        return area;
    }
}
```

Questions and Notes

What could happen in this implementation if `width`, `height` and `area` were not private?

The `this` keyword is a "magic" variable that is always present in an instance method and holds the current instance (the one that the method was called on).

Why is the code using the `this` keyword? Is there a different way to write the code that would make the `this` keyword unnecessary?

4. Interface vs. Implementation (Read pp. 324-327)

Interface

The interface of a class is its **public instance variables + public method headers + JavaDoc comments**.

The interface, or **API** (Application Programming Interface) tells the programmer everything they need to know to use your class.

Implementation

The implementation of a class is its **private stuff + method bodies + comments inside methods**.

The programmer should not have to know any of this in order to use your class.

Two Big Advantages

1. **Ease of use.** Imagine if you had to understand how `Scanner` works before you could use it!
2. **Change of Implementation.** If you keep the interface constant, you can make radical changes to the implementation of your class without breaking anybody else's code.

The Rectangle Example Again

Adapted from Listing 5.10 of the textbook…

Java

```
public class Rectangle {
    private int width;
    private int height;

    public void setDimension(int width,
                             int height) {
        this.width = width;
        this.height = height;
    }

    public int getArea() {
        return width*height;
    }
}
```

Questions and Notes

What is different about this implementation compared to the last one?

Can you think of any reason to prefer this implementation over the other or vice versa?

Would this change of implementation make any difference to a programmer who was using the Rectangle class?

Documenting your Interface (Appendix 5)

In Java, your interface must be documented in JavaDoc format. See **Documentation Standards** on Canvas or Appendix 5 of the textbook for full info. Every method gets a short description. Every parameter and return value gets a short description with `@param` and `@return`. Every class gets a short description and must have an `@author` tag.

You can use the JavaDoc compiler (from the Tools menu on IntelliJ) to generate HTML from your JavaDoc. Feel free to embed HTML tags to make the result look nicer.

EXTRA: Partial Class Diagrams for Common Classes

Diagrams like these will be made available on your tests. More to come later!