Object Oriented Programming 2 – Visuals

Digital Urban Visualization. People as Flows.

10.10.2016

iA

chirkin@arch.ethz.ch

zuend@arch.ethz.ch

treyer@arch.ethz.ch

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DARCH

iA Chair of Information Architecture

Object Oriented Programming?

Pertaining to a technique or a programming language that supports objects, classes, and inheritance.

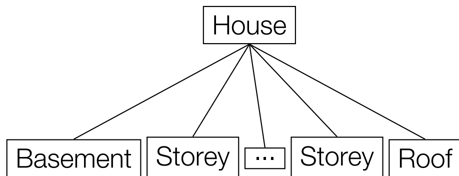
Object oriented programming languages enable a software architecture with a special data type: Object

In this lecture we will write our own objects and interactions between them.

Object composition

Today we will try to program a house, which consist of several basic elements.

We can draw each element and a house on a screen, but we are too lazy to draw all elements separatetely for each building. Therefore, we will try to compose the elements so that drawing a building will also mean drawing of all its elements.



Let's start!

```
package oop2visuals;
import processing.core.PApplet;
public class OOP2Visuals extends PApplet {
    public void setup() {
        noStroke();
        fill(180, 200, 255);
        frameRate(60);

    }
    @Override
    public void draw() {
        clear();
        background(255,255,255);
    }
    public static void main(String _args[]) {
        PApplet.main(new String[] { oop2visuals.OOP2Visuals.class.getName() });
    }
    public void settings(){
        size(800, 600);
    }
}
```

We start with our usual template.

Find it at `oop2visuals-0-template`.

This program does nothing. Yet.

Task 1 – provide a unified interface for drawing

The first thing we need to do is to decide, how should we draw everything.

This have to be some function that just gets coordinates of an element on a screen, and takes care of everything else.

Task 1 – provide a unified interface for drawing

Create new interface file `Drawable.java`

```
package oop2visuals;

import processing.core.PApplet;

public interface Drawable {

    // we will use this scale to render stuff on screen
    public static float scale = 20;

    // this is the base function to draw whatever we want to
    public void draw (PApplet canvas, float posX, float posY);
}
```

At this moment, we do not write an implementation of a function **`draw`**.

We just define it, so later we can write new classes that implement this function.

Task 1 – provide a unified interface for drawing

Create new class file `House.java`

```
package oop2visuals;

import processing.core.PApplet;
import oop2visuals.Drawable;

public class House implements Drawable{

    public House () {
    }

    @Override
    public void draw(PApplet canvas, float posX, float posY) {
    }


}
```

Now, we define a house.

At this moment function `draw` is empty, because we have no components of a building.

Task 2 – draw a basement

That was easy!.

Now, let us draw something. We can start with basement of a house.

Task 2 – draw a basement

Fill in `House.java`

```
// a base for our house
class Basement implements Drawable {

    float width = 20;
    float height = 1;
    int colourR = 10, colourG = 10, colourB = 10;

    @Override
    public void draw(PApplet canvas, float posX, float posY) {
        canvas.fill(this.colourR, this.colourG, this.colourB);
        canvas.rect( posX
                   , posY - scale*this.height
                   , scale*this.width
                   , scale*this.height);
    }


}
```

From now on, we write code into `House.java`, because all our components are rather small.

We define a `Basement` class, and explain how a program can draw it.

Task 2 – draw a basement

Update House

```
public class House implements Drawable{

    Basement base;

    public House (Basement b) {
        base = b;
    }

    @Override
    public void draw(PApplet canvas, float posX, float posY) {
        canvas.stroke(0, 0, 0);
        base.draw(canvas, posX, posY);
    }


}
```

Next thing is how to add a basement into a house.

The first line here defines a field of an object `House` – basement.

We also add something into the `draw` function: it invokes `draw` of its component.

Task 2 – draw a basement

Update `public class OOP2Visuals`

```
House house = null;

public void setup() {
    noStroke();
    fill(180, 200, 255);
    frameRate(60);

    house = new House( new Basement());
}

@Override
public void draw() {
    clear();
    background(255,255,255);
    house.draw(this, 10, 400);

}
```

The last thing: add a house to a program!

Again:

- Add a `House house` global definition;
- Setup the house at initialization time;
- Draw a house in each frame.

Try it!

Task 3 – draw house floors

Let’s try something more complicated.

Draw multiple storeys in our house!

Task 3 – draw house floors

```
// a single floor
class Storey implements Drawable{
    float width = 20;
    float height = 3;
    int colourR = 180, colourG = 80, colourB = 10;

    @Override
    public void draw(PApplet canvas, float posX, float posY) {
        // setup color
        canvas.fill(this.colourR, this.colourG, this.colourB);
        // draw floor
        canvas.rect( posX
                   , posY - scale*this.height
                   , scale*this.width
                   , scale*this.height);
        // set window color
        canvas.fill(188, 188, 211);
        // draw windows
        for (int i = 0; i < numberOfWindows(); i++) {
            canvas.rect( posX + scale*((2+windowSeperator())*i + windowSeperator()*0.5f)
                       , posY - scale*this.height*0.8f
                       , scale*2
                       , scale*this.height*0.5f);
        }
    }
    int numberOfWindows() { return Math.round((float)Math.floor(this.width / 3));}
    float windowSeperator() { return (this.width - numberOfWindows()*2) / numberOfWindows(); }
}
```

That is rather complicated code!

In fact, this is only drawing what takes so many lines of code.

This is a really simple class that implements drawable again.

A loop within `draw` stands for drawing windows.

Task 3 – draw house floors

Update `class House`

```
Basement base;
Storey[] storeys;

public House (Basement b, Storey[] ss) {
    base = b;
    storeys = ss;
}

@Override
public void draw(PApplet canvas, float posX, float posY) {
    canvas.stroke(0, 0, 0);
    float h = posY;
    base.draw(canvas, posX, h);
    h -= base.height*scale;
    for (int i = 0; i < storeys.length; i++){
        storeys[i].draw(canvas, posX, h);
        h -= storeys[i].height*scale;
    }
}
```

We need to do almost the same changes to class `House` as in the previous task.

Note here the most important detail:
when the house draws its elements, it has to calculate their positions.

Task 3 – draw house floors

Update `public class OOP2Visuals`

```
house = new House( new Basement()
                 , new Storey[] {
                       new Storey()
                     , new Storey()
                 }
                 );
```

Now, the sweet part: we almost do not need to modify our main class.

The only thing remains is to add some floors.
Try different number of floors!

Task 4 – need a door

One details is missing in our house: it does not have a door. What can we do about it?

Task 4 – add a ground floor

One details is missing in our house: it does not have a door. What can we do about it?

One solution is to modify a `Storey` class to have a door. By using inheritance!

Task 4 – add a ground floor

Update `House.java`

```
// ground floor
class GroundFloor extends Storey {

    @Override
    public void draw(PApplet canvas, float posX, float posY) {
        // first, let's draw all windows as in super class
        super.draw(canvas, posX, posY);
        // next, draw a door
        canvas.fill(100, 0, 0);
        int i = numberOfWindows() / 2;
        canvas.rect( posX + scale*((2+windowSeperator())*i + windowSeperator()*0.5f)
                   , posY - scale*this.height*0.8f
                   , scale*2
                   , scale*this.height*0.8f);
    }

}
```

We are lazy, so we do not want to draw an array of windows one more time manually.

So, let's just modify existing draw function:

1. Call a superclass function `super.draw(..);`
2. Overwrite one window with a door.

We are done!

Task 4 – add a ground floor

Update `public class OOP2Visuals`

```
house = new House( new Basement()
                 , new Storey[] {
                       new GroundFloor()
                     , new Storey()
                     , new Storey()
                 }
                 );
```

Just change the first floor to be `GroundFloor` instead of `Storey`.
Try it!

Task 5 – draw a roof

House is almost complete.

But it's missing a roof – let us fix it.

Task 5 – draw a roof

Update House.java

```
// roof - abstract, because there are different types of roofs
abstract class Roof implements Drawable{
    float width = 20;
    float height = 0.5f;
    int colourR = 0, colourG = 80, colourB = 80;

}
```

There are so many different types of roof, so I decided to make a base class abstract.

You may try to implement you own!

Task 5 – draw a roof

Update class House

```
Basement base;
Storey[] storeys;
Roof     roof;

public House (Basement b, Storey[] ss, Roof r) {
    base = b;
    storeys = ss;
    roof = r;
}

@Override
public void draw(PApplet canvas, float posX, float posY) {
    canvas.stroke(0, 0, 0);
    float h = posY;
    base.draw(canvas, posX, h);
    h -= base.height*scale;
    for (int i = 0; i < storeys.length; i++){
        storeys[i].draw(canvas, posX, h);
        h -= storeys[i].height*scale;
    }
    roof.draw(canvas, posX, h);
}
```

Here is how we add a roof to the house.

Task 5 – draw a roof

Update `public class OOP2Visuals`

```
house = new House( new Basement()
                 , new Storey[] {
                       new GroundFloor()
                     , new Storey()
                     , new Storey()
                 }
                 , new SlopedRoof()
                 );
```

...And add a roof to the house object in the main class.

Note, you cannot create an abstract roof, so you have to write your own implementation. You can find couple examples in `oop2visuals-5-roof`. Try it!

Task 6 – someone to live inside

Our house is done!

So far it all is static. But I want to add some animation to our scene. How can I do this?

Remark – state machines

One popular approach to represent behavior of something in a model – is to implement a *state machine*.

In short, it is an object or data type that has a number of different states.
For example, a state of a door may be (opened or closed); when you enter a room, the door changes its state for a while.
A person in our example will have four states:

```
enum State {AT_HOME, AT_OFFICE,
GOING_HOME, GOING_TO_OFFICE }
```

Task 6 – someone to live inside

Create a new java class `Human.java`

```
public class Human implements Drawable {

    public enum State { AT_HOME, AT_OFFICE, GOING_HOME, GOING_TO_OFFICE}
    public State state = State.GOING_HOME;
    public float stateT = 0;
    public float roadTime = 5, homeTime = 10, workTime = 8;

    ...
}
```

A person can be in one of four simple states of (enumerable) type `Human.State`.

`stateT` – time this person spent in current state;

Task 6 – someone to live inside

Create a new java class Human.java

```
@Override
public void draw(PApplet canvas, float posX, float posY) {
    float maxX = (float) canvas.width + scale*5;
    updateState(canvas);
    // render! (only on the road)
    switch (this.state) {
        case AT_HOME:
            break;
        case AT_OFFICE:
            break;
        case GOING_TO_OFFICE:
            drawMe(canvas, posX + (maxX - posX)*this.stateT, posY);
            break;
        case GOING_HOME:
            drawMe(canvas, posX + (maxX - posX)*(1f - this.stateT), posY);
            break;
    }
}
```

Function `draw` becomes quite complicated – it depends on a human state.

Note `updateState(canvas)` – a function that updates a human state.

Task 6 – someone to live inside

```
// update internal state of a human
void updateState(PApplet canvas){
    // update state timer
    switch (this.state) {
    case GOING_HOME:
        this.stateT += 1f/canvas.frameRate/this.roadTime; break;
    case GOING_TO_OFFICE:
        this.stateT += 1f/canvas.frameRate/this.roadTime; break;
    case AT_HOME:
        this.stateT += 1f/canvas.frameRate/this.homeTime; break;
    case AT_OFFICE:
        this.stateT += 1f/canvas.frameRate/this.workTime; break;
    }
    // change state if needed
    if(this.stateT >= 1)
    {
        this.stateT = 0;
        switch (this.state) {
        case GOING_HOME:
            this.state = State.AT_HOME; break;
        case GOING_TO_OFFICE:
            this.state = State.AT_OFFICE; break;
        case AT_HOME:
            this.state = State.GOING_TO_OFFICE; break;
        case AT_OFFICE:
            this.state = State.GOING_HOME; break;
        }
    }
}
```

`updateState(canvas)` – updates the time counter `this.stateT`. When `this.stateT >= 1`, a state tran sition happens.

Use the full class implementation in folder `oop2visua` `human` to run the example.

Task 6 – someone to live inside

Update `public class OOP2Visuals`

```
House house = null;
Human John = null;
Human Jane = null;

public void setup() {
    ..
    house = new House( new Basement(), new Storey[] {
                    new GroundFloor(), new Storey(), new Storey() }
                  , new SlopedRoof()
                  );
    John = new Human();
    Jane = new Human();
    Jane.workTime = 8;
    Jane.roadTime = 2;
    Jane.homeTime = 3;
}
@Override
public void draw() {
    ...
    house.draw(this, 10, 400);
    John.draw(this, 250, 400);
    Jane.draw(this, 250, 400);

}
```

This time you need to add plenty of things; namely, all inhabitants, one-by-one.

Try it!

Task 7 – smart house

Let us add some small nicies.
Smart house can turn lights on and off when its dweller comes home or leaves somewhere.

How to do such a thing? Simple!

Task 7 – smart house

Update class `Storey`

```
class Storey implements Drawable{
    ...
    public void draw(PApplet canvas, float posX, float posY) {
        ...
        // set window color
        smartTurnLightsOn(canvas);
        // draw windows
        ...
    }
    private Human owner = null;
    public void registerOwner(Human man) {
        this.owner = man;
    }
    void smartTurnLightsOn(PApplet canvas) {
        if(this.owner != null && this.owner.state == Human.State.AT_HOME) {
            canvas.fill(255, 255, 211);
        } else {
            canvas.fill(188, 188, 211);
        }
    }
```

We need to turn lights in all windows of a certain flat when its owner comes back from work. Thus, we need to update `Storey` class.

The plan:

1. Add a reference to a `Human owner`;
2. Allow registering owner;
3. Add a function `smartTurnLightsOn()` to turn lights on only when the owner is at home.

Task 7 – smart house

Update `public class OOP2Visuals`

```
setup() {
    ...
    house.storeys[1].registerOwner(John);
    house.storeys[0].registerOwner(Jane);
    ...
```

Here, we only need to register our humans as dwellers of the house.

Try it!