IEEE
Computer Chapter
Southern Alberta Section

SCHULICH
School of Engineering
UNIVERSITY OF CALGARY

IEEE Women in Engineering
WIE
Southern Alberta Section

SCHULICH IGNITE 2019

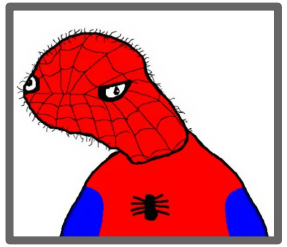
Extra Slides

- Introduction to **Classes** and **Objects** in depth
- Introduction to **Libraries**
- Other useful tips

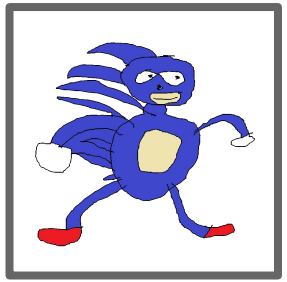
Objects

- Imagine you, an *avid* programmer, are going to make a game.
- Now, suppose there are <u>three</u> players and each has chosen different characters.
- Each character has **stats/attributes** that will be used throughout the game and may be changed.
- Where will you keep all this information?
- How will you organize it?

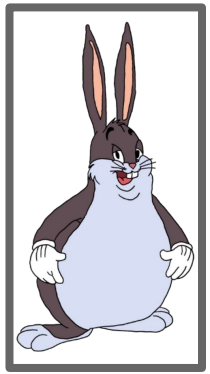
Speed: 80
Stamina : 70
Power: 50
Stealth: 100
Health: 200



Speed: 200
Stamina : 50
Power: 30
Stealth: 80
Health: 140



Speed: 20
Stamina : 40
Power: 90
Stealth: 50
Health: 300



Organization

You might just declare variables for each stat such as:

```
int sanicHealth = 140;
int sanicSpeed  = 200;
// and so on…

int chungusHealth = 300;
int chungusSpeed  =  20;

// and so on...
```

- But how would you account for characters interacting with each other?
- What if there were *more* characters?
- What if another player wanted to join?

Organization

A possible solution is to use **arrays**!

```
int[] health = {200, 140, 300};
int[] speed  = {80,  200,  20};
// and so on...
```

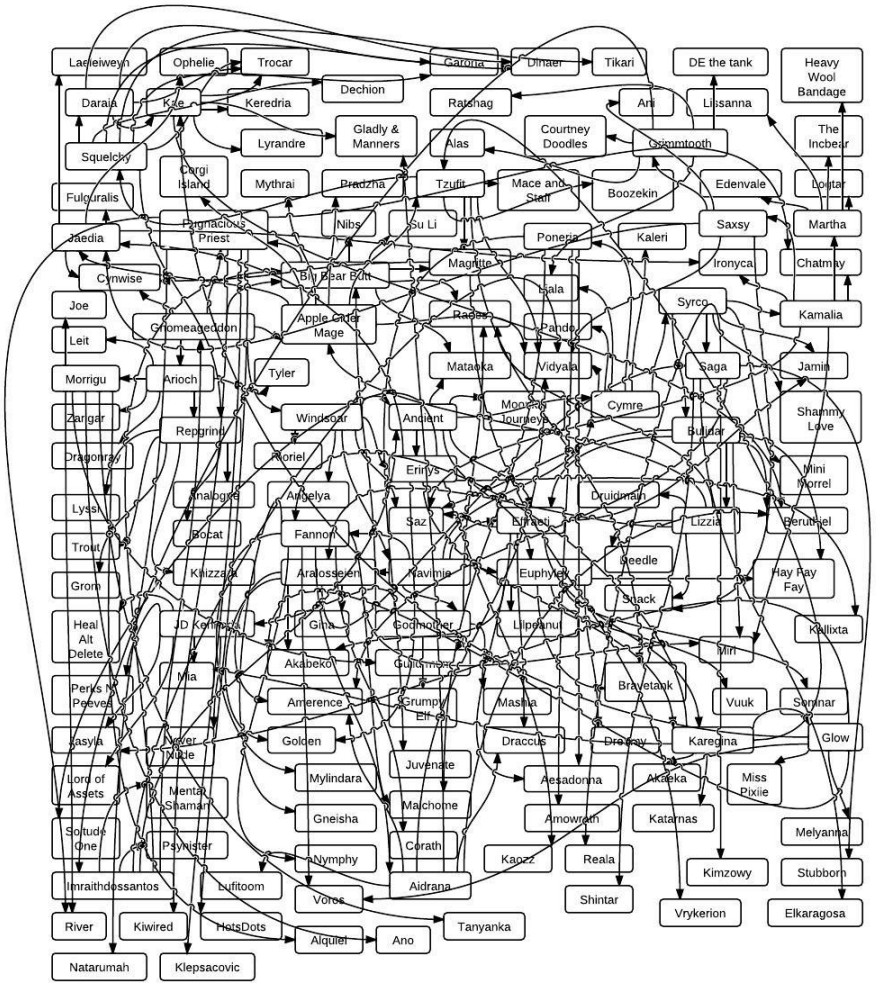
- This solves the problem of identifying characters (using their array index)
- Also allows you to add new characters (just add a new element to each array).

Disorganization

The only trouble is, this can become *very disorganized*!

You'll want to use functions to do certain things like calculate damage and check if a character is dead, and everything else.

Each function will have to call other functions which call other functions, which call other functions...



Unmanageable

- Just imagine if something went wrong or you wanted to tweak something a bit. Disaster!
- Everything depends on other things, but in ways that are not clear
- This is where **objects** become very useful

Objects

- **Objects** (also called **instances**) are a way to group together related variables and functions into separate, **modular** entities
- Essentially, they are like a superduper variable that can have other variables and functions *inside* it
- Useful because they model real-world objects
- Software engineers call this grouping, separation, and organization of code **encapsulation**
- Makes larger scale programs *much* more manageable. *Trust us.*

Objects

Using out earlier example, let's look at a better way to manage our characters using **objects**:

```
// Spoderman attacks Big Chungus
spoderman.attack(chungus);

// Sanic jumps
sanic.jump();

// Big Chungus regenerates
chungus.regenerate();
```

Objects

- Much more intuitive, right? Each action has its own line of code.
- It becomes more **human-readable**, which *always* makes things easier.
- Spoderman and everything that has to do with Spoderman is completely self-contained.
- This is the beauty of **encapsulation**.
- Designing programs based on objects is called **Object Oriented Programming** or **OOP**
- This is a *huge* topic in software engineering

```
player1Health -= 4;
```

```
damagePlayer(1, 4);
```

```
player1.damage(4);
```

How Do We Make Objects?

To be able to write clear and concise code like that, there is some design and setup involved.

We need to initialize, or **instantiate** an object (another name for an object is an **instance**):

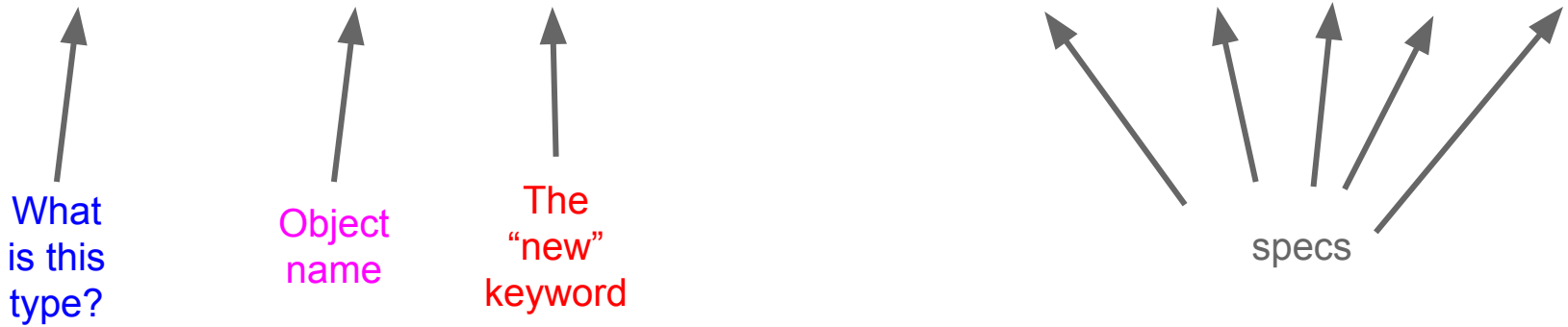
```
PlayableChar sanic = new PlayableChar(“Sanic”, 200, 50, 30, 140);
```

How Do We Make Objects?

Let's look at what's going on here.

This looks an awful lot like initializing an array, with a few differences. That's because in Processing, arrays *are* a type of object.

```
PlayableChar sanic = new PlayableChar("Sanic", 200, 50, 30, 140);
```



How Do We Make Objects?

To create an *object* is pretty straightforward, but it involves some setup too.

That variable type PlayableChar is called a **class**, and it's the reason this all works.

Classes

Classes

- Classes are a template for creating objects
- This is where the inner workings of the object are defined
- Contains functions, variables, and how they work together

To define a class:

```
class ClassName {
    // everything goes in here
}
```

What's In a Class?

Basic classes contain these three things:

1. **Fields** (variables)
2. **Methods** (functions)
3. **Constructors** (a special kind of method)

Fields

Fields

- When variables are in a class, we call them **fields** or **member variables**.
- Fields in a class do not exist until an object is instantiated
  - `boolean isAlive = 0;` sets a **default value**
- They are just instructions of what to do when you want to create an object

```
class PlayableChar {
    String name;
    int speed;
    int stamina;
    int power;
    int stealth;
    int health;
    boolean isAlive = true;
    float posX;
    float posY;

    // constructors
    // methods
}
```

Fields

Remember earlier when we said that every array has a variable called length? This is a **field** in the Array class.

If you make two arrays:

```
int[] arr1 = {1, 2, 3};
int[] arr2 = {4, 5, 6, 7, 8};
```

Recall that arr1.length is a variable equal to 3 and arr2.length is a variable equal to 5

They each have their *own* field length with independent values.

The Member Access Operator

You may have guessed this already, but the way we access fields (and methods) in an object is with the **dot operator**(“.”), also called the **member access operator**.

If I wanted to see Big Chungus’ current health, I can look inside the object and print the health field

```
println( chungus.health );
```

A field can also be *assigned to* like any other variable

```
chungus.health -= 2; // This is technically bad practice
```

Exercise 1: Fields

A class doesn't *need* to have all three elements to work.
You are allowed to make a class with only fields

1. **Create** a class called Box with three fields:
   length, width, and height.
   What is the best data type to use?
2. **Declare** a Box object called shoebox
3. **Assign** each field a value (whatever you want)
4. **Modify** each value
5. **Print** each value to the console

Methods

Methods

- When functions are in a class, we call them **methods** or **member functions.**
- Methods in a class cannot be called until an object is instantiated.
- They model real actions that the object itself might “do”
- They have direct access to the fields of the object that they are in

```
class PlayableChar {
    // fields
    // constructors

    void attack(PlayableChar victim) {
        // decrease victim's health
        // increase points or whatever
    }
    double calcDist(PlayableChar dest) {
        // find distance between
        // yourself and dest
    }
    void regenerate() {
        // increase health
    }
}
```

Methods Example

Let's take a look at what the method calcDist() would look like:

```
double calcDist(PlayableChar dest) {
    double distance;
    distance = sq(xPos - dest.xPos) + sq(yPos - dest.yPos);
    distance = sqrt(distance);
    return distance;
}
```

Methods Example

So, if we want to know how far Sanic is from Big Chungus, we can just call the calcDist() method using the **dot operator**:

```
chungus.calcDist(sanic); // this will return the distance

sanic.calcDist(chungus);
// Because of the pythagorean theorem, this is equivalent
```

Exercise 2: Methods

Now that we know about methods, let's add them to our Box class!

1. **Create** a method called volume() that calculates the volume of your box. What will your return type be? **Print** out the result.
2. **Create** a method display() that prints out the dimensions of the box.
3. **Create** a method called resize() which takes in three arguments: change in length, change in width, and change in height. Does this work for all numbers?
4. **Overload** the method resize() with one that takes one arguments and increments each dimension by that amount
5. **Show** that your functions work by printing before and after values.

Constructors

What is a Constructor?

- Finally, a good class should have a constructor.
- Constructors are *special* methods that “set up” an object when it is instantiated
- Constructors have no return value and must be the same name of the class
- Constructors can be and often are overloaded

```
class PlayableChar {
    // fields

    PlayableChar() {
        // initialize fields
        // to default
    }

    PlayableChar(/*parameters*/) {
        // initialize fields
        // to match the arguments
    }

    // methods
}
```

Constructor Example

Back when we talked about objects, we gave an example of instantiating a PlayableChar object like this:

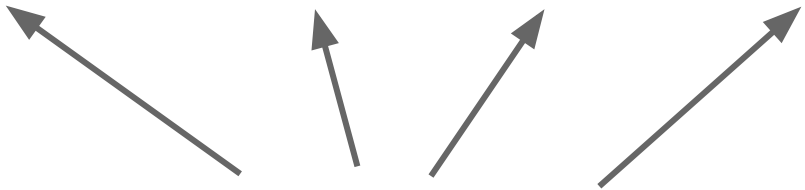
```
PlayableChar sanic = new PlayableChar(“Sanic”, 200, 50, 30, 140);
```

The highlighted portion is just calling the constructor like a regular function

Constructor Example

```
PlayableChar(String n, int sp, int stm, int pow, int h, int stlth) {
    name = n;
    speed = sp;
    stamina = stm;
    power = pow;
    stealth = stlth;
    health = h;

    isAlive = true;
    posX = 0;
    posY = 0;
}
```



Exercise 3: Constructors

To complete our Box class, we need to add constructors so that we can make a box object

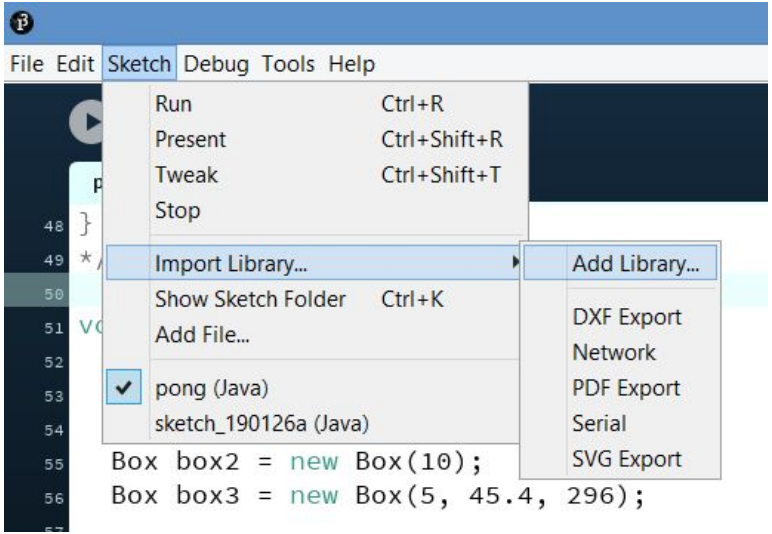
1. **Create** a constructor that takes no arguments and initializes all dimensions to 1
2. **Create** another constructor that takes one argument and initializes all dimensions to that value
3. **Create** a third constructor that takes three arguments and initializes all the dimensions respectively
4. **Declare and instantiate** three different Box objects, using each of your constructors
5. **Print** out the dimensions of each box to test that it works!

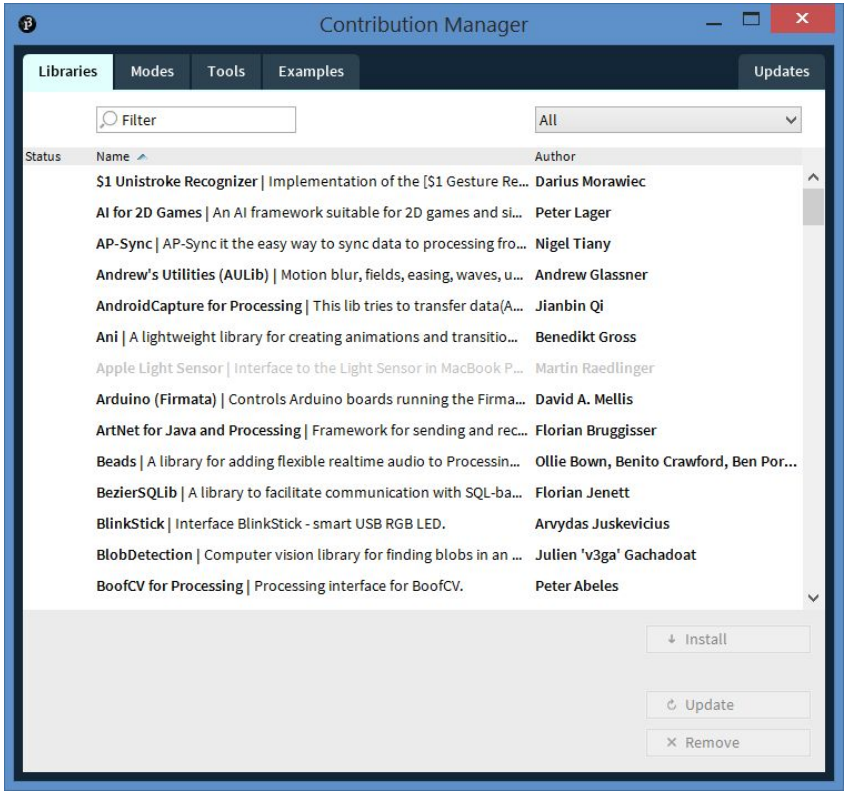
Libraries

Libraries

- A **library** is a collection of classes that other people have made for you to use
- Allows you to do things that Processing couldn't otherwise do
- You need to download them or otherwise link them to your program with an **import** statement
- Processing has built-in libraries, but you can get more on the internet or with the **Processing Contribution Manager**
- They usually come with **documentation** on how to use them

Libraries


File Edit Sketch Debug Tools Help
Run Ctrl+R
Present Ctrl+Shift+R
Tweak Ctrl+Shift+T
Stop
Import Library...
Show Sketch Folder Ctrl+K
Add File...
pong (Java)
sketch_190126a (Java)
Add Library...
DXF Export
Network
PDF Export
Serial
SVG Export
Box box2 = new Box(10);
Box box3 = new Box(5, 45.4, 296);


Contribution Manager
Libraries
Modes
Tools
Examples
Updates
Filter
All
Status
Name
Author
$1 Unistroke Recognizer | Implementation of the [$1 Gesture Re... Darius Morawiec
AI for 2D Games | An AI framework suitable for 2D games and si... Peter Lager
AP-Sync | AP-Sync it the easy way to sync data to processing fro... Nigel Tiany
Andrew's Utilities (AULib) | Motion blur, fields, easing, waves, u... Andrew Glassner
AndroidCapture for Processing | This lib tries to transfer data(A... Jianbin Qi
Ani | A lightweight library for creating animations and transitio... Benedikt Gross
Apple Light Sensor | Interface to the Light Sensor in MacBook P... Martin Raedlinger
Arduino (Firmata) | Controls Arduino boards running the Firma... David A. Mellis
ArtNet for Java and Processing | Framework for sending and rec... Florian Bruggisser
Beads | A library for adding flexible realtime audio to Processin... Ollie Bown, Benito Crawford, Ben Por...
BezierSQLib | A library to facilitate communication with SQL-ba... Florian Jenett
BlinkStick | Interface BlinkStick - smart USB RGB LED. Arvydas Juskevicius
BlobDetection | Computer vision library for finding blobs in an ... Julien 'v3ga' Gachadoat
BoofCV for Processing | Processing interface for BoofCV. Peter Abeles
Install
Update
Remove

Other Useful Tips