Classes and Objects

An introduction

Produced by:
Dr. Siobhán Drohan
Mr. Colm Dunphy
Mr. Diarmuid O'Connor

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   – Default
   – Parameters
   – Overloading

Classes and Objects

- A **class**
  - defines a group of related **methods** (functions) and **fields** (variables / properties).

Classes and **Objects**

- An **object**
  - is a single instance of a class
  - i.e. an object is created (instantiated) from a class.

String is
the Class

```
String a;
```
&FFCC

&FFCC
"Hello"

a is the Object,
which contains "Hello"
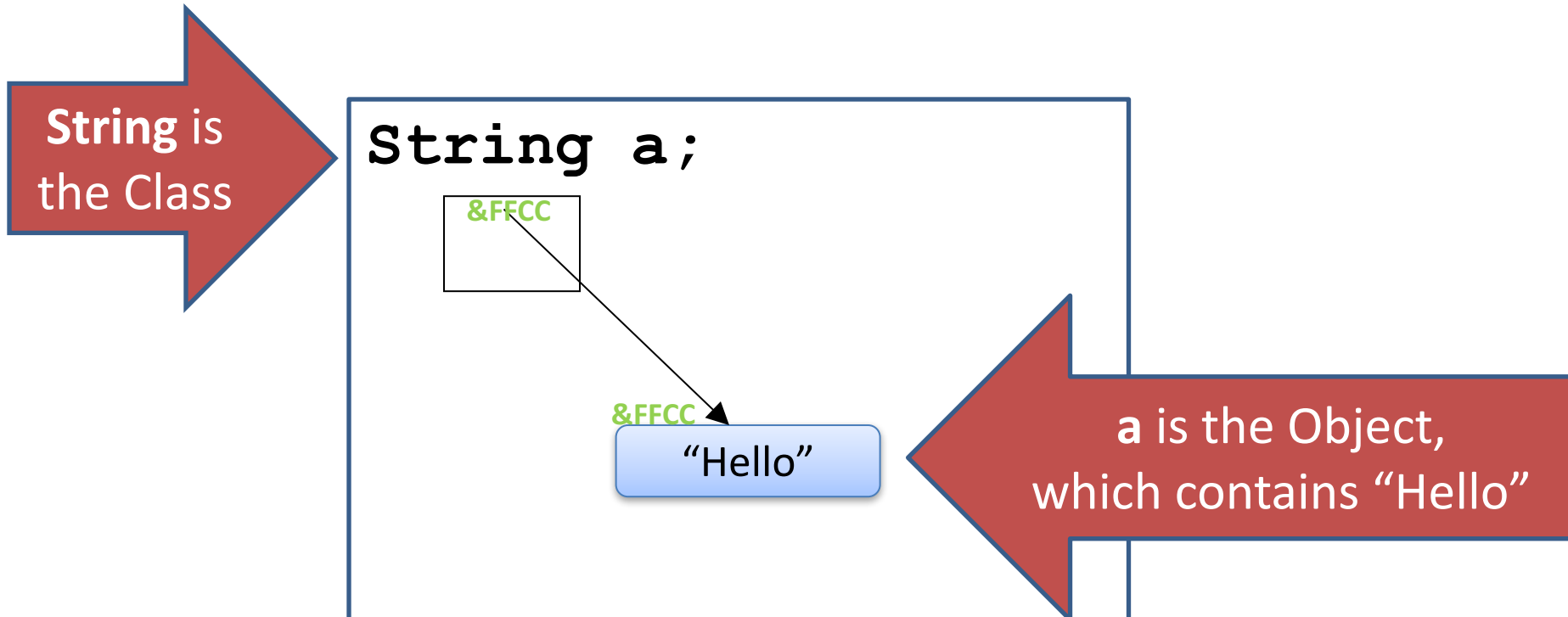
Classes and Objects – **1) Building Analogy**

- ## A **class** is like a **blueprint** for a building.
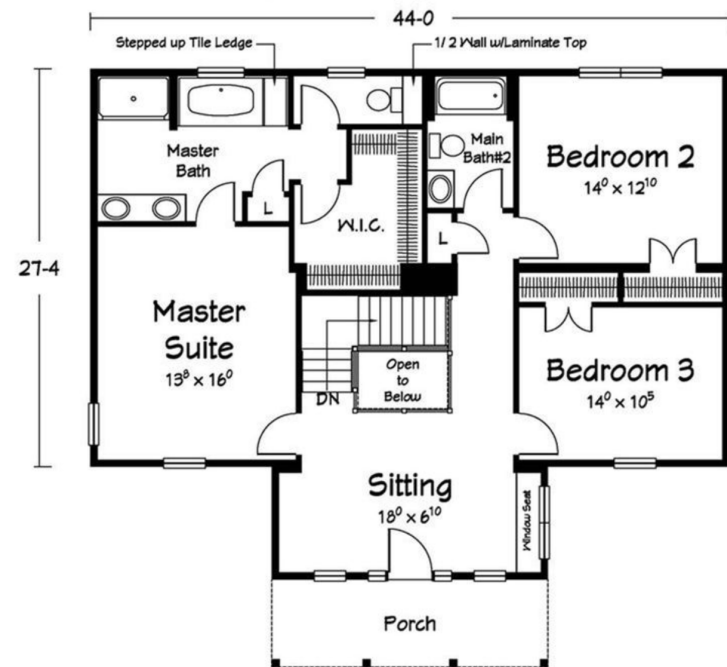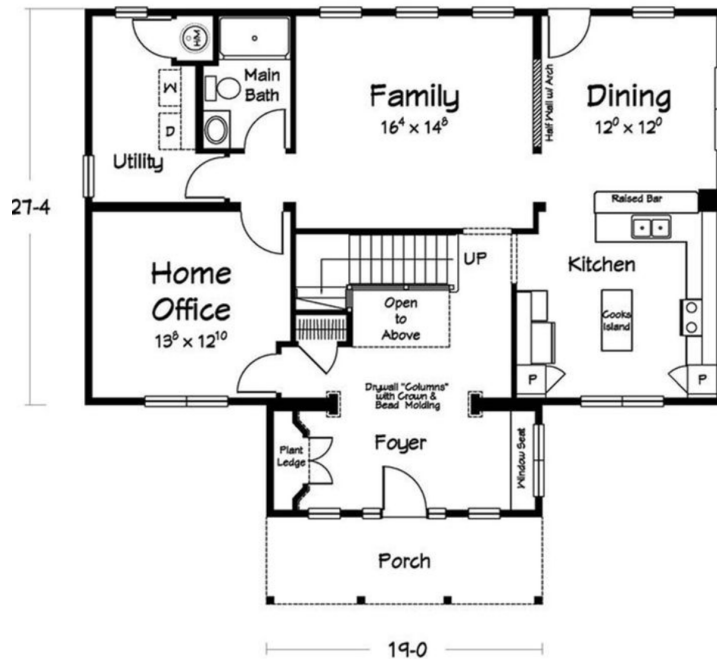
Classes and Objects – 1) Building Analogy

- An **object** is a **building** constructed from that blueprint.
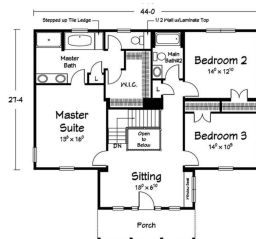
Classes and Objects – 1) Building Analogy

- You can build lots of (buildings) **objects** from a single blueprint.

Classes and Objects – **2) Cake Analogy**

- A **class** is like a **recipe** for a cake.

What you need:

- 175g/6oz Odlums Cream Plain Flour
- 75g/3oz Plain Chocolate (min 70% cocoa)
- 200g/7oz Butter
- 175g/6oz Shamrock Golden Caster Sugar
- 3 Large Eggs
- 1 teaspoon Baking Powder
- 100g packet Shamrock Ground Almonds
- 2 tablespoons Cocoa, sieved
- 2 tablespoons Milk
- 1 teaspoon Goodall's Vanilla Essence

For Chocolate Cream

- 140ml Cream
- 175g/6oz Plain Chocolate (min 70% cocoa)

How to:

1. Preheat oven to 190°C/375°F/Gas 5. Lightly grease and base line a 23cm/9" deep sandwich tin.

2. Break the chocolate into a heatproof bowl. Add 25g/1oz of the butter and stand bowl over a pan of hot water until chocolate has melted.

3. Meanwhile put the remaining butter, sugar, eggs, flour, baking powder, ground almonds, cocoa, milk and essence into a large bowl and beat until smooth and creamy.

4. Add the melted chocolate and gently stir into the mixture. Transfer to the prepared tin and level the top.

5. Bake for about 40 minutes until risen and the surface feels firm to the touch. Remove from oven. Allow to sit in tin for about 5 minutes, then transfer to a wire tray to cool.

6. Make the chocolate cream by heating the cream until just bubbling around the edges. Add the chocolate and gently stir over a low heat until melted. Remove from heat.

7. Transfer to a bowl to allow to cool and begin to set.

8. Slice cake horizontally and use half the icing to sandwich the cake.

9. Spread remaining icing on top and sides of cake.

10. Serve with raspberries and crème fraîche or Greek yoghurt.

Classes and Objects – 2) Cake Analogy

- An **object** is the **cake** baked from that recipe.

Classes and Objects – 2) Cake Analogy

- You can bake **lots of** (**cakes**) **objects** from a single recipe.

Classes and Objects – Many Objects

- Many **objects** can be constructed from a single **class** definition.

- Each **object** must have a **unique name** within the program.

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   - Default
   - Parameters
   - Overloading

Methods (functions) and **Fields** (variables/properties)

- Objects are typically related to real-world artefacts.

- In object-oriented programming (e.g. Java), you model an object by grouping together related **methods** (functions) and **fields** (variables).

Object example: **Apple**
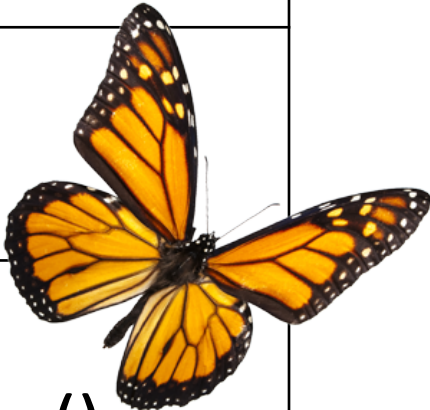
Object Name	**Apple**
Fields (variables, properties)	color weight
Methods (functions)	grow() fall() rot()

Object example: **Butterfly**

Object Name	Butterfly
Fields (variables, properties)	species gender
Methods (functions)	grow() flapWings() land()

Object example: **Radio**

Object Name	Radio
Fields (variables, properties)	frequency volume
Methods (functions)	turnOn() tune() setVolume()

Object example: **Car**

Object Name	Car
Fields (variables, properties)	make model color year
Methods (functions)	accelerate() brake() turn()

Returning to the Apple Example

Object Name	Apple
Fields (variables, properties)	color weight
Methods (functions)	grow() fall() rot()

Returning to the Apple Example

Object Name	Apple
Fields (variables, properties)	color weight
Methods (functions)	grow() fall() rot()

Object Type

Returning to the Apple Example

Object Name	Apple
Fields (variables, properties)	**color** **weight**
Methods (functions)	grow() fall() rot()

Properties / Attributes

Returning to the Apple Example

Object Name	Apple
Fields (variables, properties)	color weight
Methods (functions)	grow() fall() rot()

Behaviours

Apple Class

- To make a " blue print" of an Apple:

- The **grow()** method
  - might have inputs for temperature and moisture.
  - can increase the **weight** field of the apple based on these inputs.

Apple
color *weight*
grow() *fall()* *rot()*

Apple Class

- To make a " blue print" of an Apple:

- The **fall()** method
  - can continually check the **weight** and cause the apple to fall to the ground when the weight goes above a threshold.

Apple
color *weight*
grow() **fall()** *rot()*

Apple Class

- To make a " blue print" of an Apple:

- The **rot()** method could then take over,
  - beginning to decrease the value of the **weight** field
  - and change the **color** fields.

Apple
color *weight*
grow() *fall()* **rot()**

Source: Reas & Fry (2014)
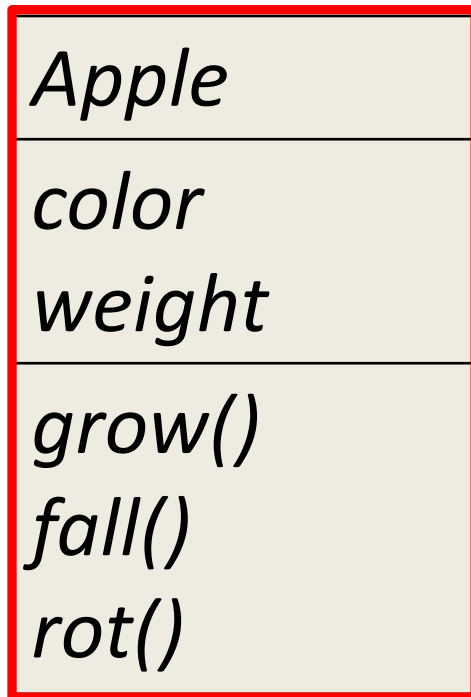
Apple Object(s)

- We saw earlier that:
  - An **object**
    - is created (**instantiated**) from a class.
  - A **class**
    - can have **many objects created from it**.
  - Each object
    - must have a **unique name** within the program.

Apple Object(s)

Apple
color
weight
grow()
fall()
rot()

Class

gala

color	red
weight	6.2

goldenDelicious

color	yellow
weight	8.4
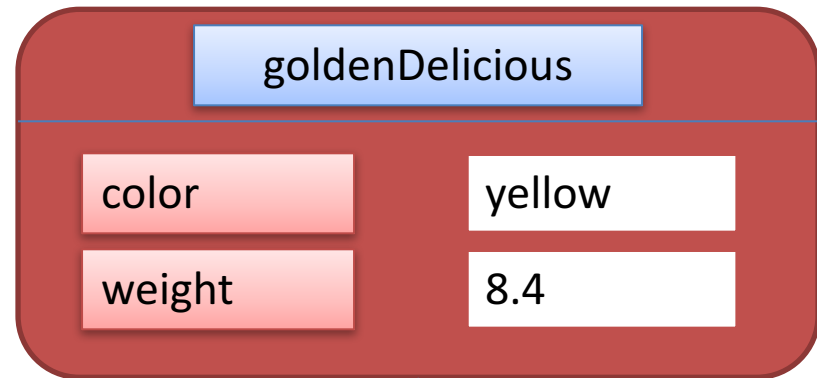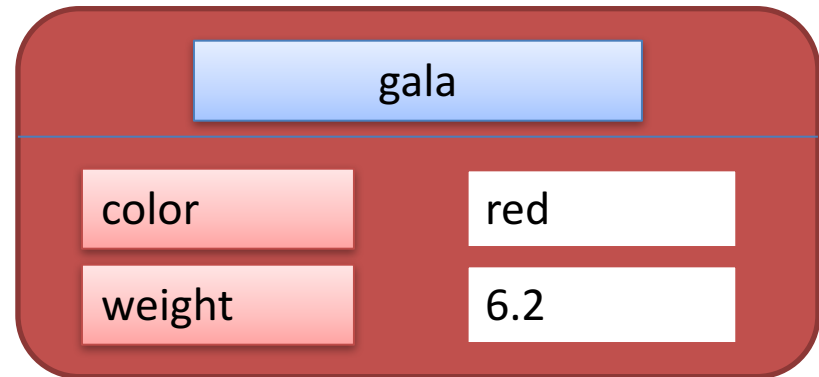
Two objects. Each has a unique name and it's own copy (values) of the fields.

Object State

There are two objects of type Apple.

Each has a unique name:
   gala
   goldenDelicious

Each object has a different **object state**:
   each object has it's own copy of the fields (color and weight) in memory and has it's own data stored in these fields.
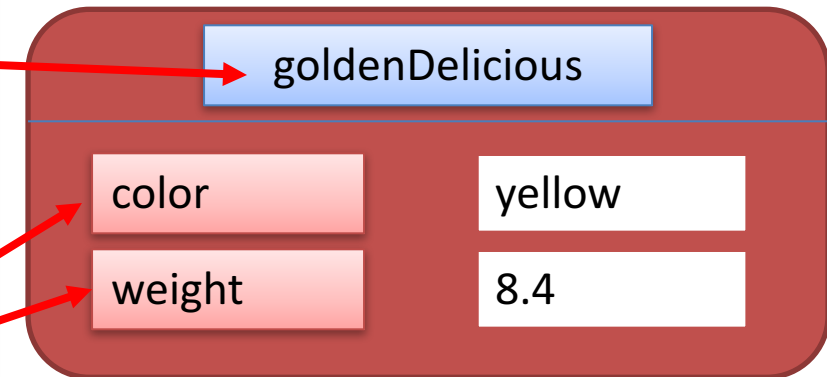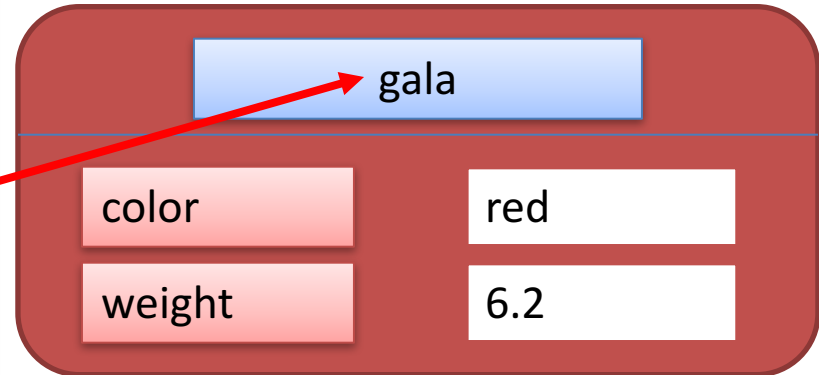
gala	
color	red
weight	6.2

goldenDelicious	
color	yellow
weight	8.4

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   – Default
   – Parameters
   – Overloading

Using an Object's fields and methods

- The fields and methods of an object are accessed with the **dot operator** i.e. external calls.

| gala**.color**	Gives access to the color value in the gala object.
goldenDelicious**.color**	Gives access to the color value in the goldenDelicious object.
gala**.grow()**	Runs the grow() method inside the gala object.
goldenDelicious**.fall()**	Runs the fall() method inside the goldenDelicious object.

object.property
object.method

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   – Default
   – Parameters
   – Overloading

Creating your first class

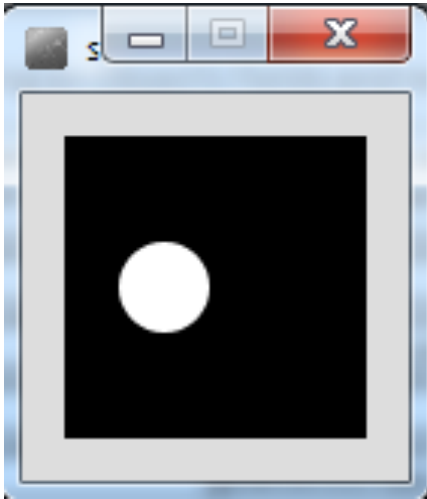- We are going to start with sample code that **draws a white spot on a black background**.

- We will **refactor** this code by:
  - writing a **class**
    - that will draw and format this spot.
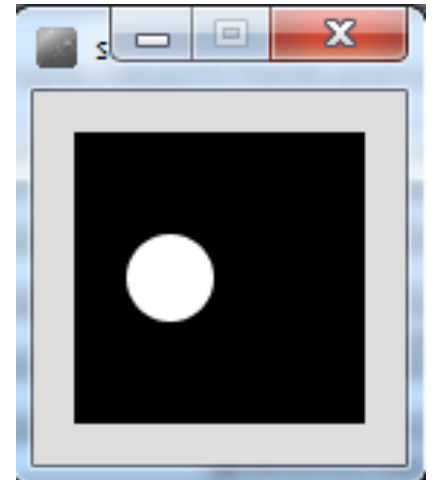
Sample Code


```
float xCoord = 33.0;
float yCoord = 50.0;
float diameter = 30.0;

void setup(){
  size (100,100);
  noStroke();
}


void draw(){
  background(0);
  ellipse(xCoord, yCoord, diameter, diameter);
}
```

Creating your first class

- A class creates a **unique data type**.

- When creating a class, think carefully about what you want the code to do:

  1. What are the **attributes**?

  2. What are the **behaviours**?

First, we will start by:
listing the **attributes** (fields/variables/properties) and figure out what **data type** they should be.

Creating your first class – identifying the **fields** (attributes, properties)

```
float xCoord = 33.0;
float yCoord = 50.0;
float diameter = 30.0;

void setup(){
  size (100,100);
  noStroke();
}

void draw(){
  background(0);
  ellipse(xCoord, yCoord, diameter, diameter);
}
```
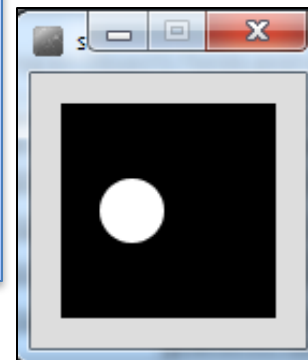
Q: What fields do we need to model the spot?

Note:

fields are the attributes/properties of the object we are modelling.

Creating your first class – identifying the fields

```
float xCoord = 33.0;
float yCoord = 50.0;
float diameter = 30.0;

void setup(){
  size (100,100);
  noStroke();
}


void draw(){
  background(0);
  ellipse(xCoord, yCoord, diameter, diameter);
}
```
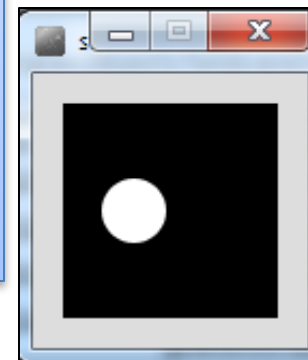
A: The required fields (attributes) are:

float **xCoord** *(x-coordinate of spot)*

float **yCoord** *(y-coordinate of spot)*
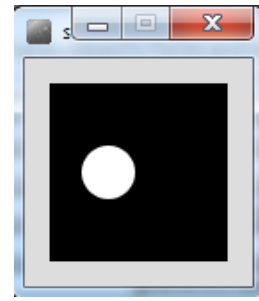
float **diameter** *(diameter of the spot)*

Creating your first class –
giving our new **class** a **name**

- The name of a class should be carefully considered and should **match its purpose**.
- The name can be any word or words.
- It should **begin with a capital letter**
- It should **not be pluralised**.
- For our first class, we could use names like:
  - Spot
  - Dot
  - Circle, etc.
- We will call our first class, **Spot**.

Spot Class – Version 1.0

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}


void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord, sp.diameter, sp.diameter);
}
```

```
class Spot
{

  float xCoord, yCoord;
  float diameter;

}
```

Spot Class – Version 1.0

Defining the **class**

```
class Spot
{
    float xCoord, yCoord;
    float diameter;
}
```

Declaring the **fields** in the class

File Edit Sketch Debug Tools Help

Spot_Version1_0   **Spot**

```
1  class Spot
2  {
3      float xCoord, yCoord;
4      float diameter;
5  }
```

In the PDE, place this code in a new **tab**, called Spot
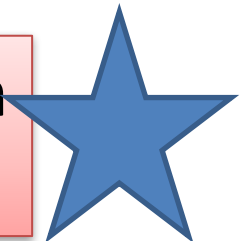
Spot Class – Version 1.0
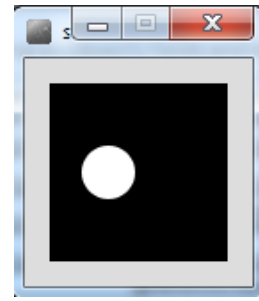
Declaring an object **sp**, of type **Spot**.
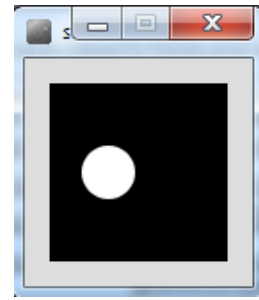
```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
              sp.diameter, sp.diameter);
}
```

```
class Spot
{
  float xCoord, yCoord;
  float diameter;
}
```

Spot Class – Version 1.0

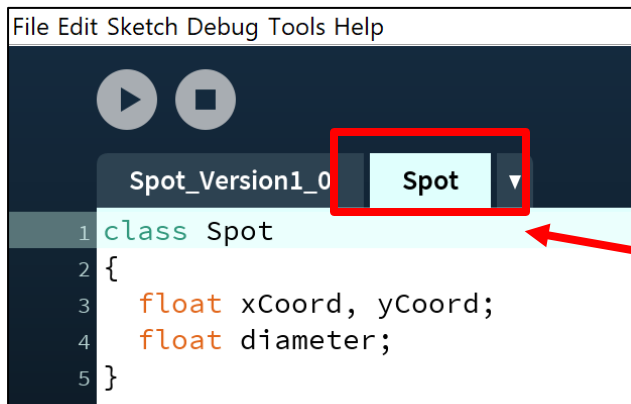Declaring an object **sp**, of type **Spot**.

Calling the **Spot()** *constructor* to build the **sp** object in memory.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
              sp.diameter, sp.diameter);
}
```

```
class Spot
{
  float xCoord, yCoord;
  float diameter;
}
```

Spot Class – Version 1.0

Declaring an object **sp**, of type **Spot**.

Calling the **Spot()** *constructor* to build the **sp** object in memory.

Initialising the fields in the **sp** object with a starting value.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}


void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
            sp.diameter, sp.diameter);
}
```
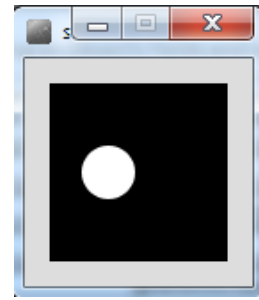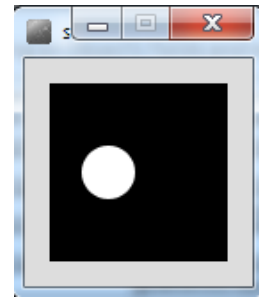
```
class Spot
{
  float xCoord, yCoord;
  float diameter;
}
```

Spot Class – Version 1.0
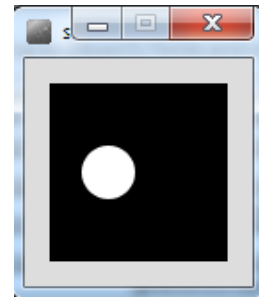
Declaring an object **sp**, of type **Spot**.

Calling the **Spot()** *constructor* to build the **sp** object in memory.

Initialising the fields in the **sp** object with a starting value.

Calling the ellipse method, **using the fields** in the **sp** object as arguments.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
          sp.diameter, sp.diameter);
}
```

```
class Spot
{
  float xCoord, yCoord;
  float diameter;
}
```

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors** ⬅
   – Default
   – Parameters
   – Overloading

Constructors

Spot sp;

sp

null

Constructors

Spot sp;

sp = new **Spot()**;

sp

> null

sp

&FFCC

&FFCC

sp

xCoord	0.0
yCoord	0.0
diameter	0.0

Constructors

```
Spot sp;
sp = new Spot();
```

The sp object is **constructed** with the keyword **new**.

Spot() is the *default constructor* that is called to build the **sp** object in memory.

A **CONSTRUCTOR** is a method that has the **same name as the class** but has **no return type**.

```
Spot()
{

}
```

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   ➡ – Default
   – Parameters
   – Overloading

Default Constructor

```
class Spot
{
    float xCoord;
    float yCoord;
    float diameter;

    //Default Constructor
    Spot()
    {
    }
}
```

The default constructor has an empty parameter list.

Default Constructor

```
class Spot
{
    float xCoord;
    float yCoord;
    float diameter;

    //Default Constructor
    Spot()
    {
    }
}
```

- If you don't include a constructor in your class, the compiler inserts a default one for you in the background

  (i.e. you won't see it in your code).

Default Constructor

```
class Spot
{
    float xCoord;
    float yCoord;
    float diameter;

    //Default Constructor
    Spot()
    {
    }
}
```
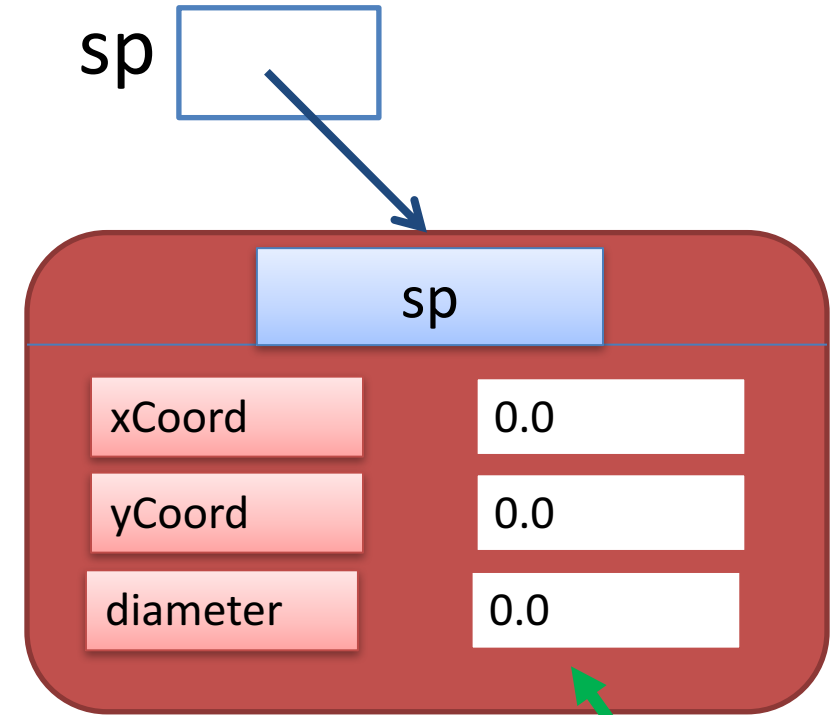
Here, the Spot()
default constructor
simply constructs the object.

Default Constructor

```
class Spot
{
    float xCoord;
    float yCoord;
    float diameter;

    //Default Constructor
    Spot()
    {
    }
}
```

sp

sp

xCoord	0.0
yCoord	0.0
diameter	0.0

The constructor stores **initial values** in the fields.

Writing our first constructor

We now know that **constructors store initial values in the fields of the object**:

- They often receive external parameter values for this.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
          sp.diameter, sp.diameter);
}
```

Writing our first constructor

In this code, we initialized:

- xCoord
- yCoord
- diameter

after calling the Spot() constructor.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
              sp.diameter, sp.diameter);
}
```

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   – Default
   ➡ – Parameters
   – Overloading

Writing our first constructor

We want to write a new constructor that will take three parameters

- xPos
- yPos
- diamtr

These values will be used to initialise the

- xCoord,
- yCoord
- Diameter

variables.

```
Spot sp;

void setup(){
  size (100,100);
  noStroke();
  sp = new Spot();
  sp.xCoord = 33;
  sp.yCoord = 50;
  sp.diameter = 30;
}

void draw(){
  background(0);
  ellipse(sp.xCoord, sp.yCoord,
                sp.diameter, sp.diameter);
}
```

Writing our first constructor

We want to write a new constructor that will take three parameters

- xPos
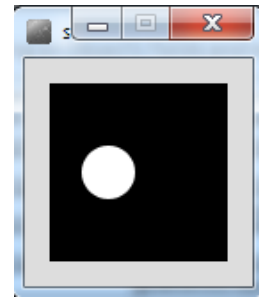- yPos
- diamtr

These values will be used to initialise the

- xCoord,
- yCoord
- Diameter

variables.

```
class Spot
{
  float xCoord, yCoord;
  float diameter;

  Spot(float xPos, float yPos, float diamtr)
  {
    xCoord = xPos;
    yCoord = yPos;
    diameter = diamtr;
  }
}
```

Spot Class – Version 2.0

```
Spot sp;

void setup()
{
  size (100,100);
  noStroke();
  sp = new Spot(33, 50, 30);
}

void draw()
{
  background(0);
  ellipse(sp.xCoord, sp.yCoord, sp.diameter, sp.diameter);
}
```

```
class Spot
{
  float xCoord, yCoord;
  float diameter;

  Spot(float xPos, float yPos, float diamtr)
  {
    xCoord = xPos;
    yCoord = yPos;
    diameter = diamtr;
  }
}
```

Source: Reas & Fry (2014)

TOPICS

1. Classes & Objects
2. **Properties** (fields, variables, attributes) & **Methods** (functions)
3. **Dot**
4. Creating your first class – **Spot**
5. **Constructors**
   – Default
   – Parameters
   ➡ – Overloading

Overloading Constructors

- We can have as many constructors as our design requires, ONCE they have unique parameter lists.

- We are overloading our constructors in Version 3.0…
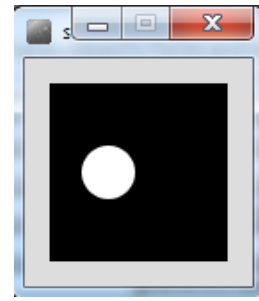
Spot Class – Version 3.0
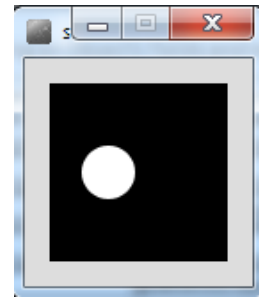
```
Spot sp;

void setup()
{
  size (100,100);
  noStroke();
  sp = new Spot(33, 50, 30);
}

void draw()
{
  background(0);
  ellipse(sp.xCoord, sp.yCoord, sp.diameter, sp.diameter);
}
```

```
class Spot{
  float xCoord, yCoord;
  float diameter;

  Spot(){
  }

  Spot(float xPos, float yPos, float diamtr){
    xCoord = xPos;
    yCoord = yPos;
    diameter = diamtr;
  }
}
```
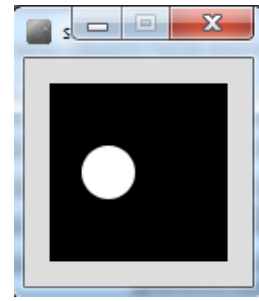
Source: Reas & Fry (2014)

Spot Class – Version 3.0

Default Constructor with NO parameters

```
class Spot{
  float xCoord, yCoord;
  float diameter;

  Spot(){
  }

  Spot(float xPos, float yPos, float diamtr){
    xCoord = xPos;
    yCoord = yPos;
    diameter = diamtr;
  }
}
```

Spot Class – Version 3.0

A second Constructor with a (float, float, float) parameter list.

```
class Spot{
  float xCoord, yCoord;
  float diameter;

  Spot(){
  }

  Spot(float xPos, float yPos, float diamtr){
    xCoord = xPos;
    yCoord = yPos;
    diameter = diamtr;
  }
}
```

Questions?

References

- Reas, C. & Fry, B. (2014) Processing – A Programming Handbook for Visual Designers and Artists, 2nd Edition, MIT Press, London.