



# Programming Fundamentals 1

---

Produced  
by

Mr. Dave Drohan  
([david.drohan@setu.ie](mailto:david.drohan@setu.ie))

Dr. Siobhán Drohan

Ms. Mairead Meagher  
Department of Computing & Mathematics  
South East Technological University  
Waterford, Ireland

setu.ie





# Introduction to Processing

## Classes and Objects

### Classes and Objects



objects · classes · Spot



# Agenda

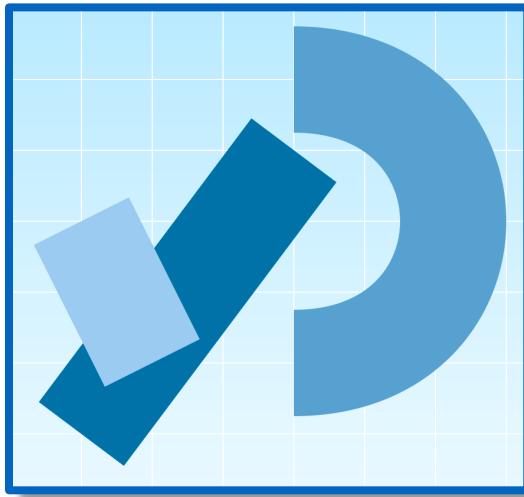
---

- Classes & Objects
- Properties & Methods
- The **Dot** Operator
- Creating your first class – **Spot**
- Constructors
  - Default
  - Parameters
  - Overloading



---

# Classes and Objects





# Classes and Objects

## ☐ A class

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

The screenshot shows the Java API documentation for the `String` class. The URL is <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>. The page title is "String (Java Platform SE 7)". The navigation bar includes links for Overview, Package, Class (which is highlighted), Use, Tree, Deprecated, Index, and Help. Below the navigation bar are links for Prev Class, Next Class, Frames, No Frames, and All Classes. The main content area displays the `java.lang` package and the **Class String**. It also lists inheritance from `java.lang.Object` and `java.lang.String`.

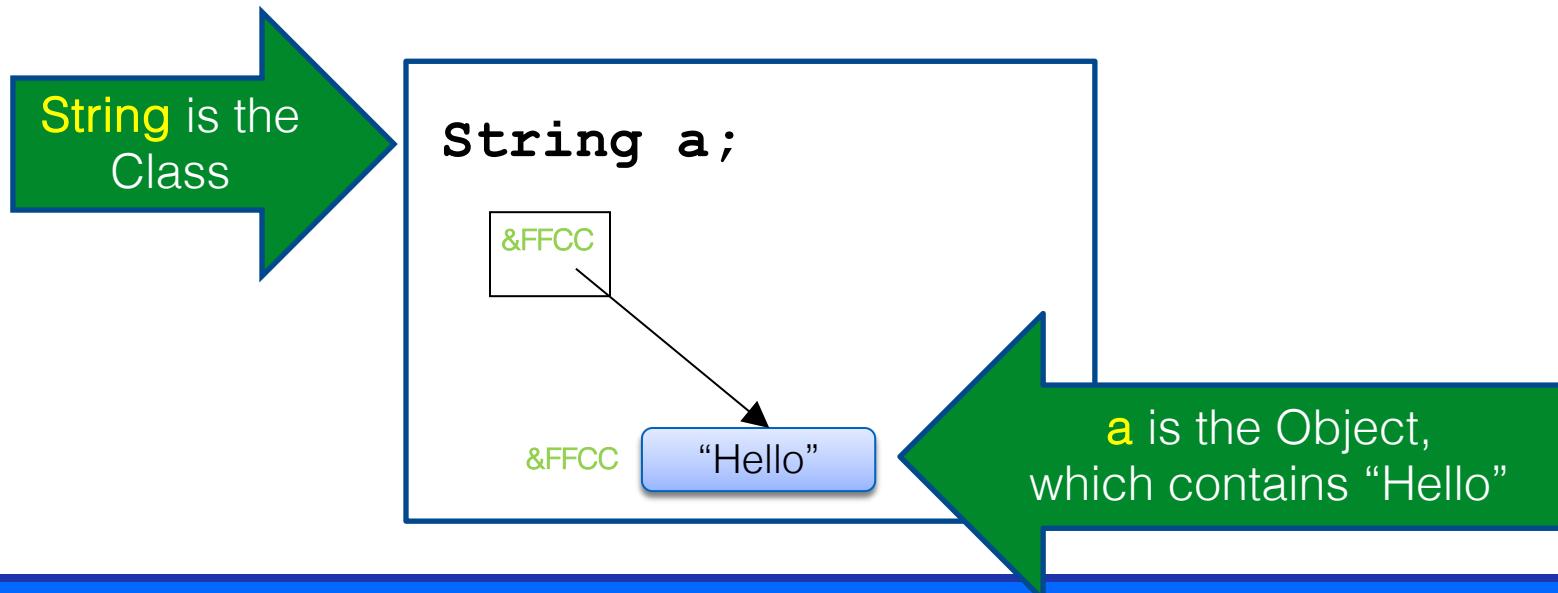
Method Summary	
Methods	Method and Description
char	<code>charAt(int index)</code> Returns the char value at the specified index.
int	<code>codePointAt(int index)</code> Returns the character (Unicode code point) at the specified index.
int	<code>codePointBefore(int index)</code> Returns the character (Unicode code point) before the specified index.
int	<code>codePointCount(int beginIndex, int endIndex)</code> Returns the number of Unicode code points in the specified text range of this <code>String</code> .
int	<code>compareTo(String anotherString)</code> Compares two strings lexicographically.
int	<code>compareToIgnoreCase(String str)</code> Compares two strings lexicographically, ignoring case differences.
String	<code>concat(String str)</code> Concatenates the specified string to the end of this string.
boolean	<code>contains(CharSequence s)</code> Returns true if and only if this string contains the specified sequence of char values.
boolean	<code>contentEquals(CharSequence cs)</code> Compares this string to the specified CharSequence.
boolean	<code>contentEquals(StringBuffer sb)</code> Compares this string to the specified StringBuffer.
static String	<code>copyValueOf(char[] data)</code> Returns a <code>String</code> that represents the character sequence in the array specified.
static String	<code>copyValueOf(char[] data, int offset, int count)</code> Returns a <code>String</code> that represents the character sequence in the array specified.
boolean	<code>endsWith(String suffix)</code> Tests if this string ends with the specified suffix.
boolean	<code>equals(Object anObject)</code> Compares this string to the specified object.



# Classes and Objects

## ❑ An **object**

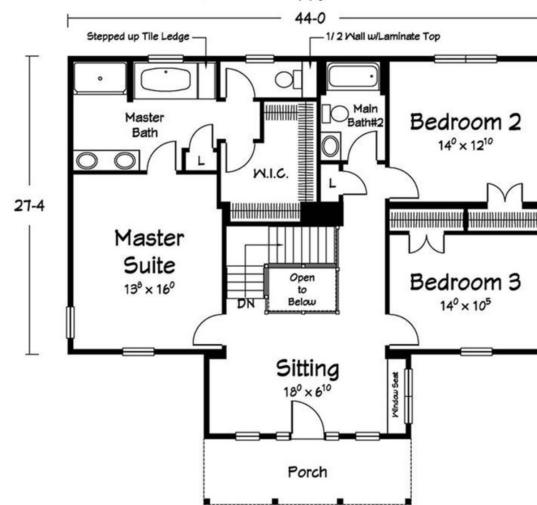
- is a single instance of a class
- i.e. an object is created (instantiated) from a class.





# Classes and Objects – Building Analogy

- A **class** is like a blueprint for a building.





# Classes and Objects – Building Analogy

- An **object** is a building constructed from that blueprint.





# Classes and Objects – Building Analogy

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





# Classes and Objects – Many Objects

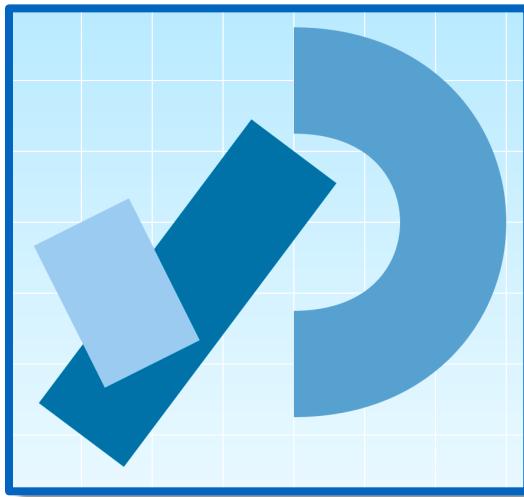
---

- Many **objects** can be constructed from a single **class** definition.
- Each **object** must have a **unique name** within the program.



---

# Properties & Methods





## Methods (functions) & 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)	colour 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 colour year
Methods (functions)	accelerate() brake() turn()





# Returning to the Apple Example

---

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





# Returning to the Apple Example

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

Object Type



# Returning to the Apple Example

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

Properties /  
Attributes



# Returning to the Apple Example

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





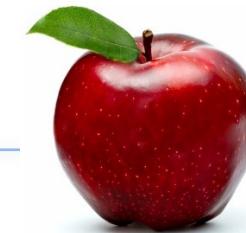
# Apple Class

□ 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.

<i>Apple</i>
<i>colour</i>
<i>weight</i>
<i>grow()</i>
<i>fall()</i>
<i>rot()</i>





# 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.



<i>Apple</i>
<i>colour</i>
<i>weight</i>
<code>grow()</code>
<code>fall()</code>
<code>rot()</code>



# 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 **colour** fields.

<i>Apple</i>
<i>colour</i>
<i>weight</i>
<i>grow()</i>
<i>fall()</i>
<i>rot()</i>



# Apple Object(s)

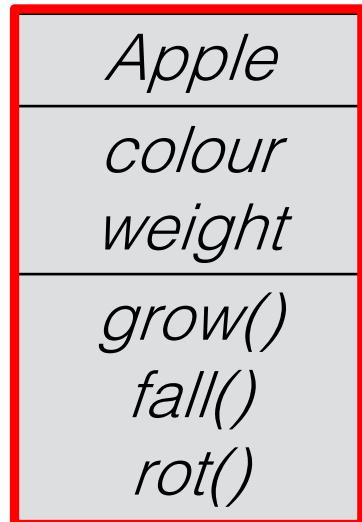
---

□ We saw earlier that:

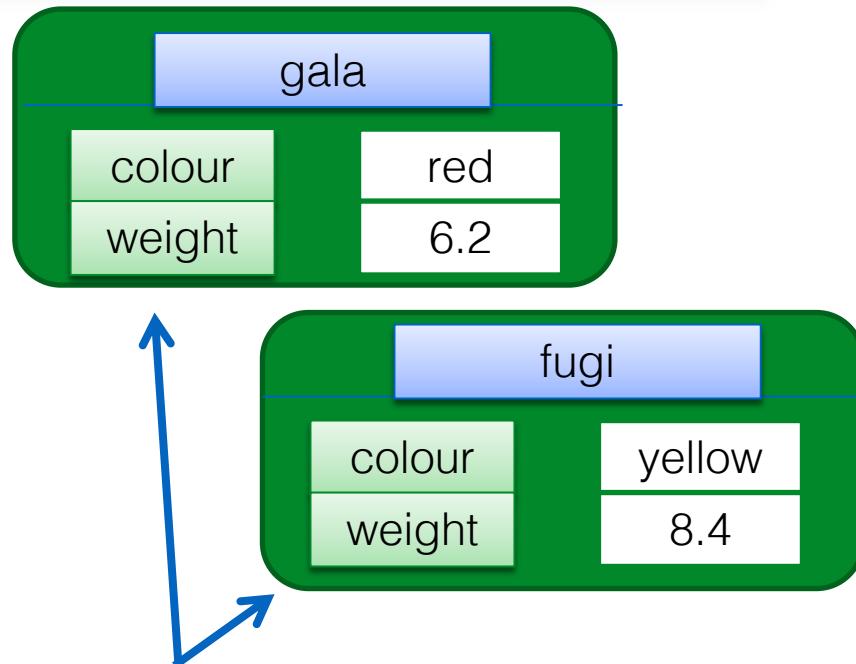
- An **object**
  - ◆ is created (**instantiated**) from a class.
- A **class**
  - ◆ can have many **objects** created from it.
- Each **object**
  - ◆ must have a **unique name** within the program.



# Apple Object(s)



Class



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



# Object State

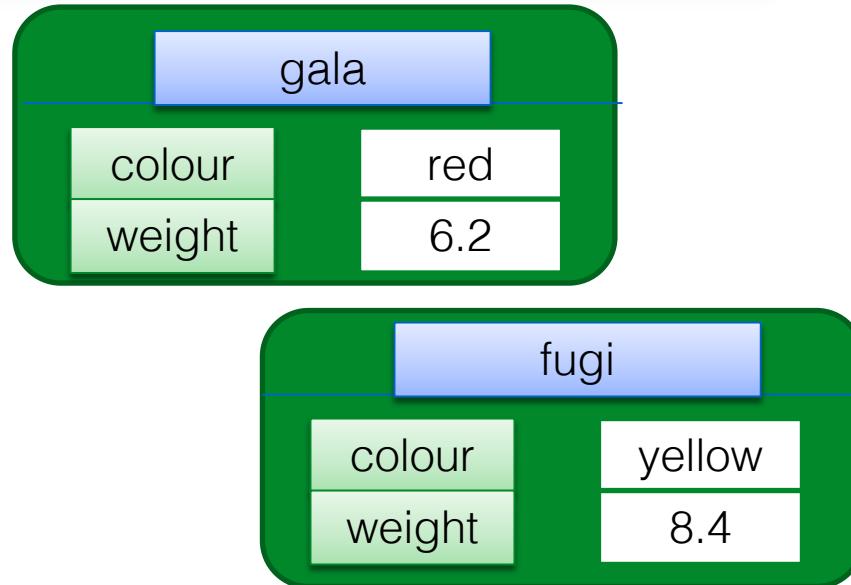
There are two **objects** of type **Apple**.

Each has a unique name:

gala  
fugi

Each object has a different **object state**:

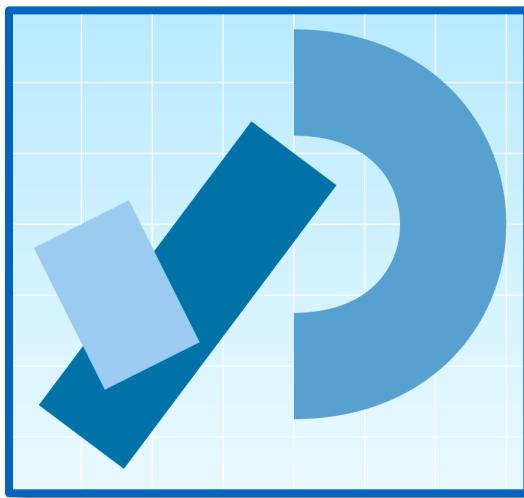
each object has it's own copy of the fields (colour and weight) in memory and has it's own data stored in these fields.





---

# The Dot Operator





# Using an Object's fields and methods

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

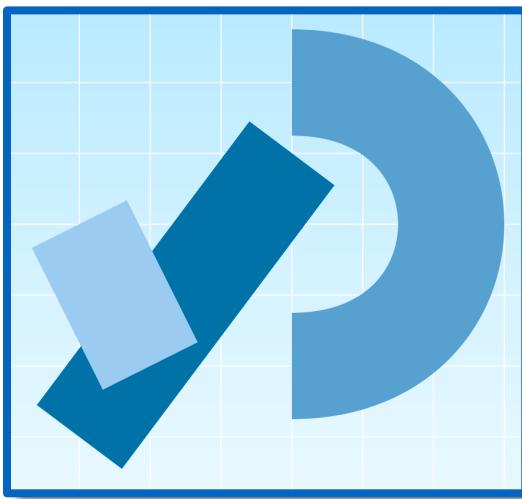
object.property  
object.method

FIELDS	gala.colour	Gives access to the colour value in the gala object.
	fugi.colour	Gives access to the colour value in the fugi object.
METHODS	gala.grow()	Runs the grow() method inside the gala object.
	fugi.fall()	Runs the fall() method inside the fugi object.



---

# Creating your first Class - Spot



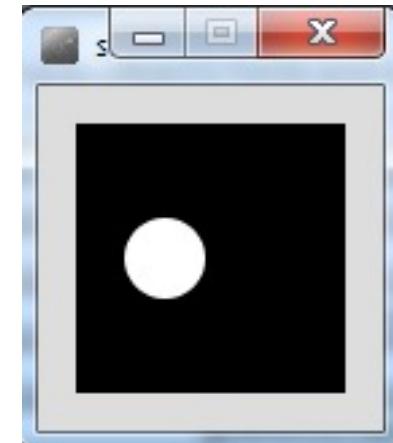


# Creating your first class

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

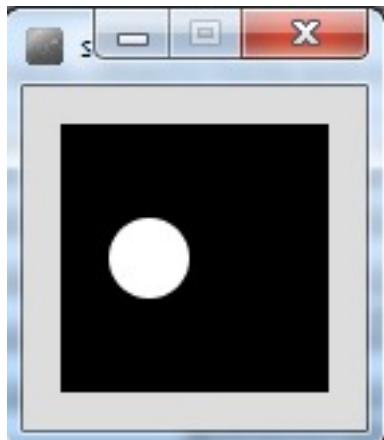
□ We will **refactor** this code by:

- 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.



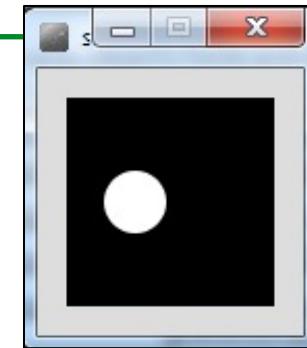
# Identifying the fields (attributes, properties)

```
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.*





# Identifying the fields (attributes, properties)

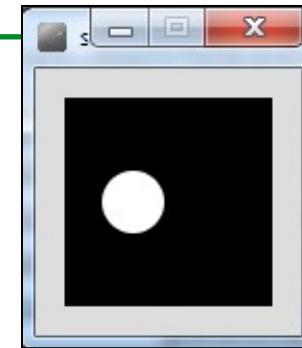
```
float xCoord = 33.0;  
float yCoord = 50.0;  
float diameter = 30.0;  
  
void setup(){  
    size (100,100);  
    noStroke();  
}  
  
void draw(){  
    background(0);  
    ellipse(xCoord, yCoord, diameter, diameter);  
}
```

A: The required fields (attributes) are:

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

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

float **diameter** (*diameter of the spot*)





# Giving our new class a name

---

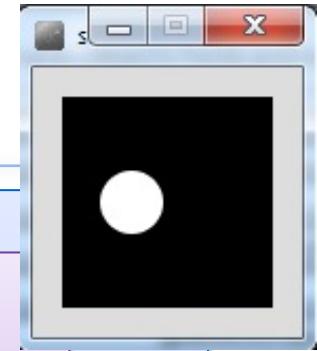
- The name of a class should be carefully considered and should **match its purpose**
- The name can be **any** word or words
- It should **begin with a capital letter**
- It should **not be pluralised**
- For our first class, we could use names like:
  - Spot
  - Dot
  - Circle, etc.
- We will call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

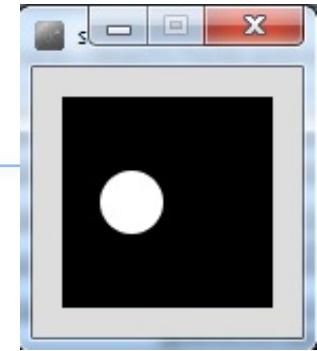


Defining the class



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

Declaring the fields in  
the class



```
sketch_230720  
sketch 230720a Spot  
1 class Spot  
2 {  
3     float xCoord, yCoord;  
4     float diameter;  
5 }  
6 |
```

In the PDE, place this code in a  
new tab, called Spot

# Spot Class – Version 1.0



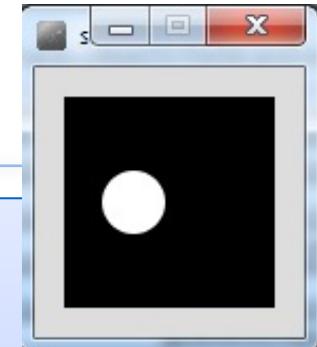
*Declaring* an object **sp**,  
of type **Spot**.

```
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**.

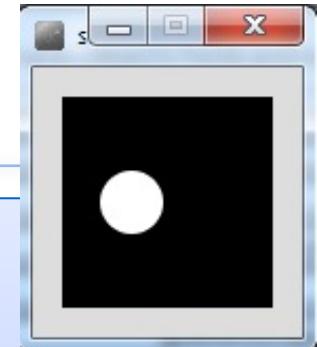
Calling the **Spot()**  
*constructor* to build the  
**sp** object in memory.

```
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



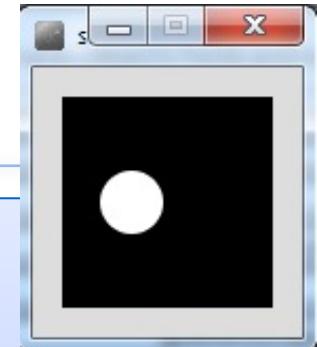
*Declaring* an object **sp**,  
of type **Spot**.

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

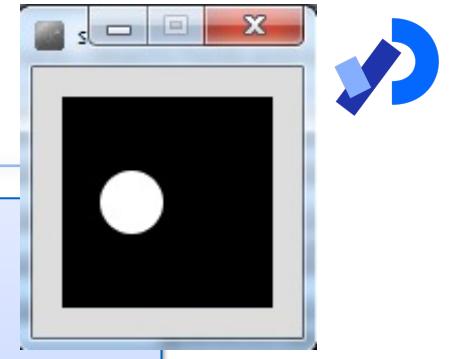
Initialising the fields in  
the **sp** object with a  
starting value.

```
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.

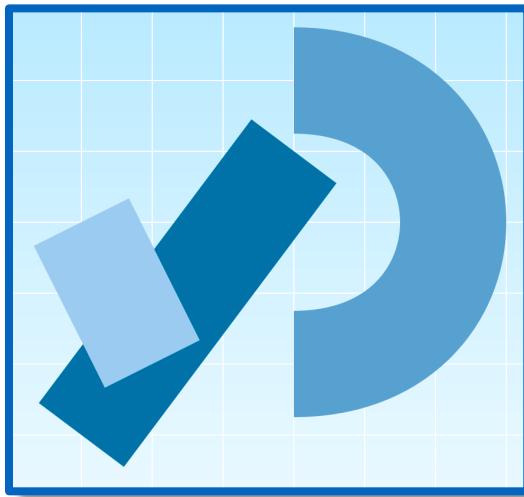
```
Spot sp;  
  
void setup(){  
    size (100,100);  
    noStroke();  
    sp = new Spot();  
    sp.xCoord = 33;  
    sp.yCoord = 50;  
    sp.diameter = 30;  
}  
  
void draw(){  
    background(0);  
    ellipse(sp.xCoord, sp.yCoord,  
        sp.diameter, sp.diameter);  
}
```

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



---

# Constructors





# Constructors

```
Spot sp;
```

sp  
  null

Declares an **sp** object variable initialised to null by default

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

sp  
  &FFC  
    C

**new** calls the constructor to allocate the object in memory and initialise it's fields





# 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()  
{  
}
```



# 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()
    {
    }
}
```

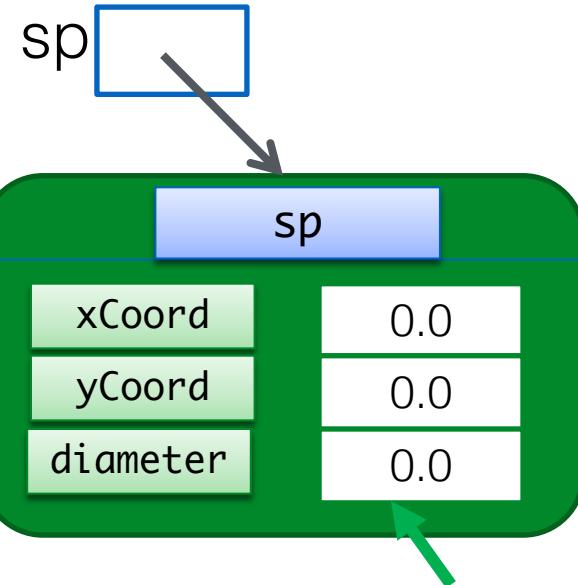
Here, the Spot() default constructor simply constructs the object.



# Default Constructor

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

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



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);  
}
```



# 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;

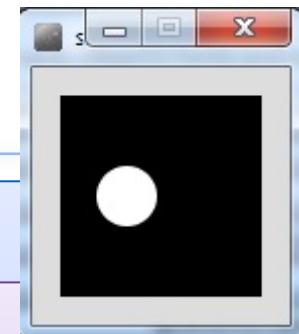
    Spot(float xPos, float yPos, float
diamtr)
    {
        xCoord = xPos;
        yCoord = yPos;
        diameter = diamtr;
    }
}
```

# 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;  
    }  
}
```





# Overloading Constructors

---

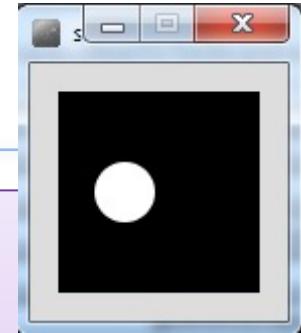
- We can have as many constructors as our design requires, **ONCE** they have unique parameter lists.
- We are **overloading** our constructors in Version 3.0...

# 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);  
}
```

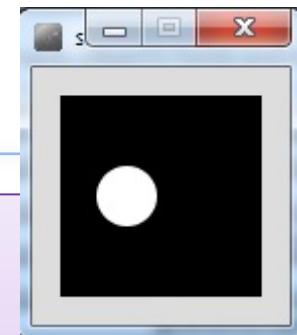
```
class Spot{  
    float xCoord, yCoord;  
    float diameter;  
  
    Spot(){  
    }  
  
    Spot(float xPos, float yPos, float diamtr){  
        xCoord = xPos;  
        yCoord = yPos;  
        diameter = diamtr;  
    }  
}
```



# 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;  
    }  
}
```

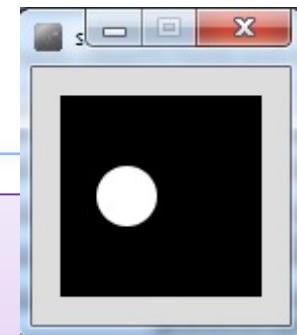


# Spot Class – Version 3.0



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, 2<sup>nd</sup> Edition, MIT Press, London.



Thanks.

