

More on Classes

Encapsulation

Produced Ms. Mairead Meagher
by: Dr. Siobhán Drohan



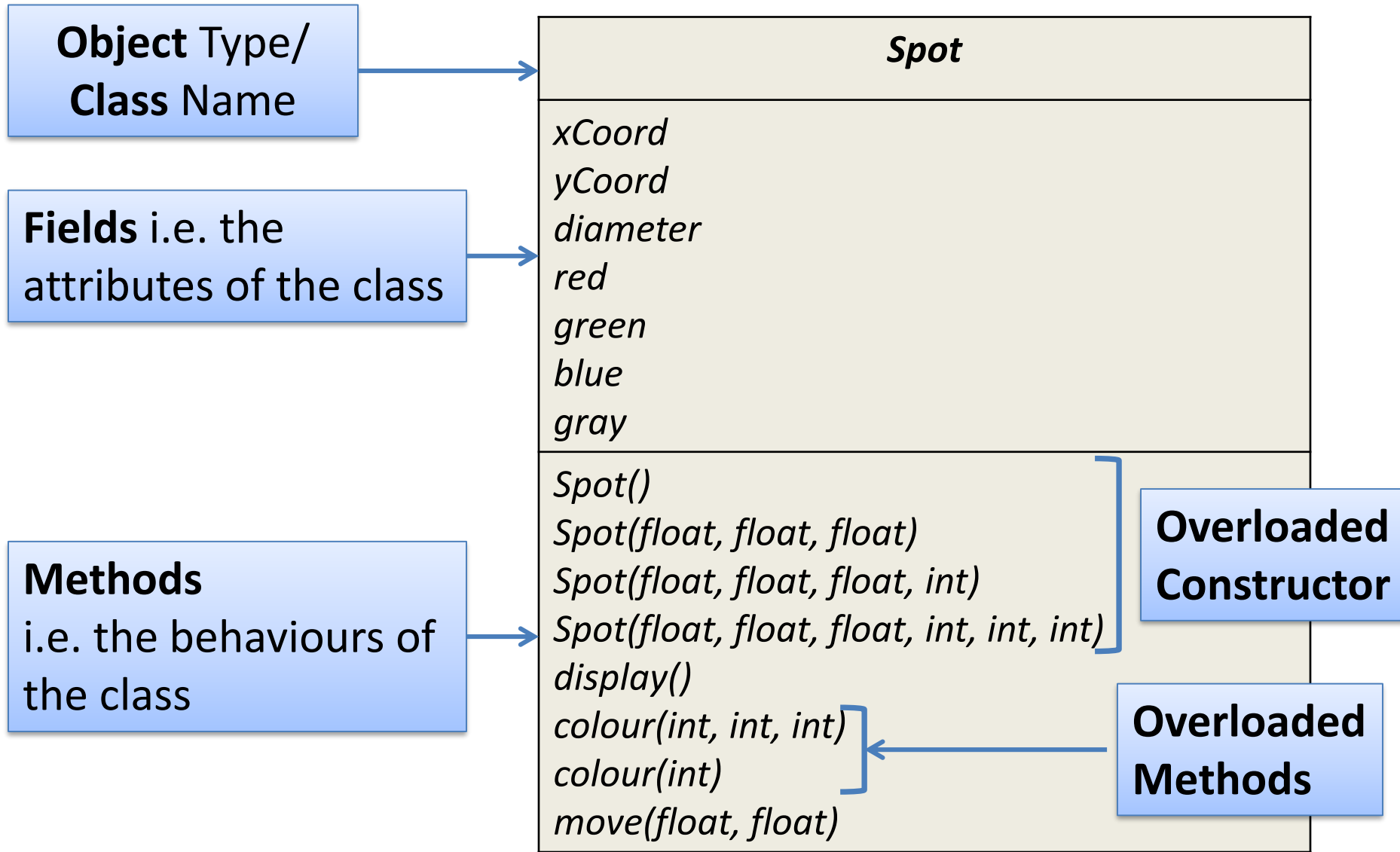
Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
<http://www.wit.ie/>

Topics list

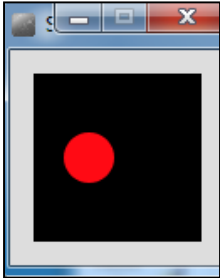
1. Recap: Version 6.1
2. Our design smells!
3. **Encapsulation**
4. Refactoring Spot:
 - Access Modifiers
 - Accessors and Mutators
 - Validation

Class Diagram for Spot Version 6.1



Spot Class

– Version 6.1



```
class Spot{  
    float xCoord, yCoord;  
    float diameter;  
    int red, green, blue;
```

```
Spot()  
{  
}
```

```
Spot (float xCoord, float yCoord, float diameter)  
{  
    this.xCoord = xCoord;  
    this.yCoord = yCoord;  
    this.diameter = diameter;  
}
```

```
// colour methods...
```

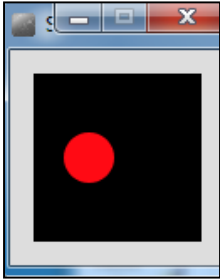
```
// display method...
```

```
// move method...
```

```
}
```

Spot Class

– Version 6.1



```
class Spot{
// fields and constructors...

void display ()
{
    ellipse(xCoord, yCoord, diameter, diameter);
}

void colour (int red, int green, int blue)
{
    this.red = red;
    this.green = green;
    this.blue = blue;
    fill (red, green, blue);
}

void colour (int gray){
    this.gray = gray;
    fill (this.gray);
}
}
```

Spot Class – Version 7.0

```
Spot sp;
```

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

```
void draw()  
{  
  background(0);  
  sp.colour(255, 0, 0);  
  sp.diameter = 30000;  
  sp.display();  
}
```

```
class Spot{  
  float xCoord, yCoord;  
  float diameter;  
  int red, green, blue;  
  
  // constructors...  
  void display(){  
    ellipse(xCoord, yCoord, diameter, diameter);  
  }  
  
  void colour(int red, int green, int blue)  
  {  
    this.red = red;  
    this.green = green;  
    this.blue = blue;  
    fill (red, green, blue);  
  }  
  
  // move methods...  
}
```

Topics list

1. Recap: Version 6.1

2. Our design smells!

3. **Encapsulation**

4. Refactoring Spot:

- Access Modifiers
- Accessors and Mutators
- Validation

Our Our design smells!

- We can directly access the diameter field (and all other fields) in the Spot class from another class, and set it to a value that is completely preposterous!
- Also, when we directly access a field in a class, we are applying a “**behaviour**” to that field i.e. resizing the circle.
 - But, aren’t **methods** supposed to be the “behaviour” for a class???????

Our Our design smells!

- Our design violates one of the basic principles of object-oriented design:

Encapsulation!

Topics list

1. Recap: Version 6.1
2. Our design smells!
- 3. Encapsulation**
4. Refactoring Spot:
 - Access Modifiers
 - Accessors and Mutators
 - Validation

Encapsulation

- **Encapsulation** (data hiding) is a fundamental Object Oriented concept.
- How to achieve encapsulation?
 1. **wrap** the data (fields) and code acting on the data (methods) together as single unit.
 2. **hide** the fields from other classes.
 3. **access** the fields only through the methods of their current class.

Encapsulation in Java – steps 1-3

Encapsulation Step	Approach in Java
1. Wrap the data (fields) and code acting on the data (methods) together as single unit.	<pre>public class <i>ClassName</i> { <i>Fields</i> <i>Constructors</i> <i>Methods</i> }</pre>
2. Hide the fields from other classes.	Declare the fields of a class as <u>private</u>.
3. Access the fields only through the methods of their current class.	Provide <u>public</u> setter and getter methods to modify and view the fields values.


Topics list

1. Recap: Version 6.1

2. Our design smells!

3. Encapsulation

4. Refactoring Spot:

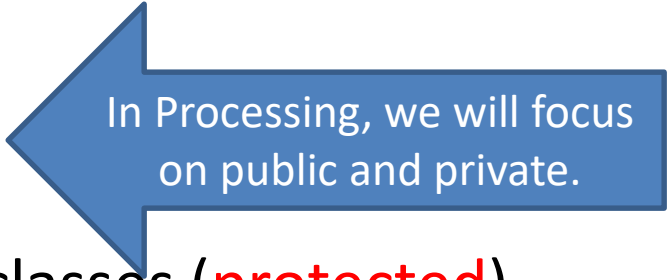
- 
- **Access Modifiers**
 - **Accessors and Mutators**
 - **Validation**

Refactoring Spot: Access Modifiers

- Java provides a number of access modifiers to set access levels for classes, fields, methods and constructors.
- The **four access levels** are:
 - Visible to the **package**, the default. No modifiers needed.
 - Visible to the class only (**private**).
 - Visible to the world (**public**).
 - Visible to the package and all subclasses (**protected**).

Refactoring Spot: Access Modifiers

- Java provides a number of access modifiers to set access levels for classes, fields, methods and constructors.
- The four access levels are:
 - Visible to the **package**, the default. No modifiers needed.
 - **Visible to the class only (**private**)**.
 - **Visible to the world (**public**)**.
 - Visible to the package and all subclasses (**protected**).



In Processing, we will focus on public and private.

Refactoring Spot 7.0: Access Modifiers

```
public class Spot{  
    float xCoord, yCoord;  
    float diameter;  
    int red, green, blue;
```

```
    Spot()  
    {  
    }
```

```
    // other constructor
```

```
    void display(){  
        ellipse(xCoord, yCoord, diameter, diameter);  
    }
```

```
    // move method...
```

```
    // colour methods...
```

```
}
```

Encapsulation step 1 is complete;
all fields, constructors and methods
are all in a single unit, called Spot.

We just changed the class access level to **public**
(default is **package**).

Filename: Spot

Refactoring Spot 7.0: Access Modifiers

```
public class Spot{  
    float xCoord, yCoord;  
    float diameter;  
    int red, green, blue;
```

```
    Spot()  
    {  
    }
```

```
    // other constructor  
    void display(){  
        ellipse(xCoord, yCoord, diameter, diameter);  
    }  
    // move method...  
    // colour methods...  
}
```

However, as the default access level is **package**
→ our methods and fields are all **package** level access.

Problem: this breaks **Encapsulation step 2**
i.e. the fields of a class should be **private**.

Refactoring Spot 7.0: Access Modifiers

```
public class Spot{
```

```
    private float xCoord, yCoord;  
    private float diameter;  
    private int red, green, blue;
```

```
    Spot()  
    {  
    }
```

```
    // other constructor
```

```
    void display(){  
        ellipse(xCoord, yCoord, diameter, diameter);  
    }
```

```
    // move method...
```

```
    // colour methods...
```

```
}
```

To fix **Encapsulation step 2**,
we declare all the fields with **private** access.

Refactoring Spot 7.0: Access Modifiers

PROBLEM: You have a garden and it is **public**. Anyone can take the properties of the garden when they want.

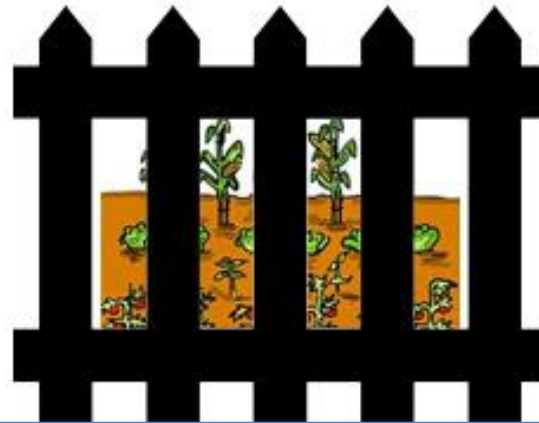


Refactoring Spot 7.0: Access Modifiers

PROBLEM: You have a garden and it is public. Anyone can take the properties of the garden when they want.



SOLUTION? Put a high fence around my garden, now it is safe! But waite, **I can no longer access my own garden.**

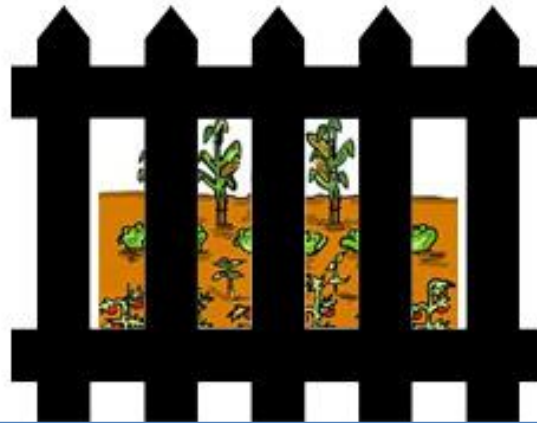


Refactoring Spot 7.0: Access Modifiers


```
public class Spot{  
    private float xCoord, yCoord;  
    private float diameter;  
    private int red, green, blue;  
    //constructors...  
    //display method...  
    // move method...  
    // colour methods...  
}
```

The **private** fields are not viewable or updatable outside the class **Spot**. Other classes don't know these exist.

SOLUTION? Put a high fence around my garden, now it is safe! But waite, **I can no longer access my own garden.**



Topics list

1. Recap: Version 6.1
2. Our design smells!
3. **Encapsulation**
4. Refactoring Spot:
 - Access Modifiers
 -  – Accessors and Mutators (getters & setters)
 - Validation

Refactoring Spot 7.0: Setters and Getters

SOLUTION: Hire a **private** guard and give him **rules** on who is able to access the garden. Anyone wanting to use the garden must get permission from guard. garden is now **safe** and **accessible**.



Refactoring Spot 7.0: Setters and Getters

SOLUTION: Hire a **private** guard and give him **rules** on who is able to access the garden. Anyone wanting to use the garden must get permission from guard. garden is now **safe** and **accessible**.



Setters and Getters to Safeguard Data



Refactoring Spot 7.0: Setters and Getters

SOLUTION: Hire a private guard and give him rules on who is able to access the garden. Anyone wanting to use the garden must get permission from guard. garden is now **safe** and **accessible**.



Encapsulation Step 3:
Provide public setter and getter methods to modify and view the fields values.

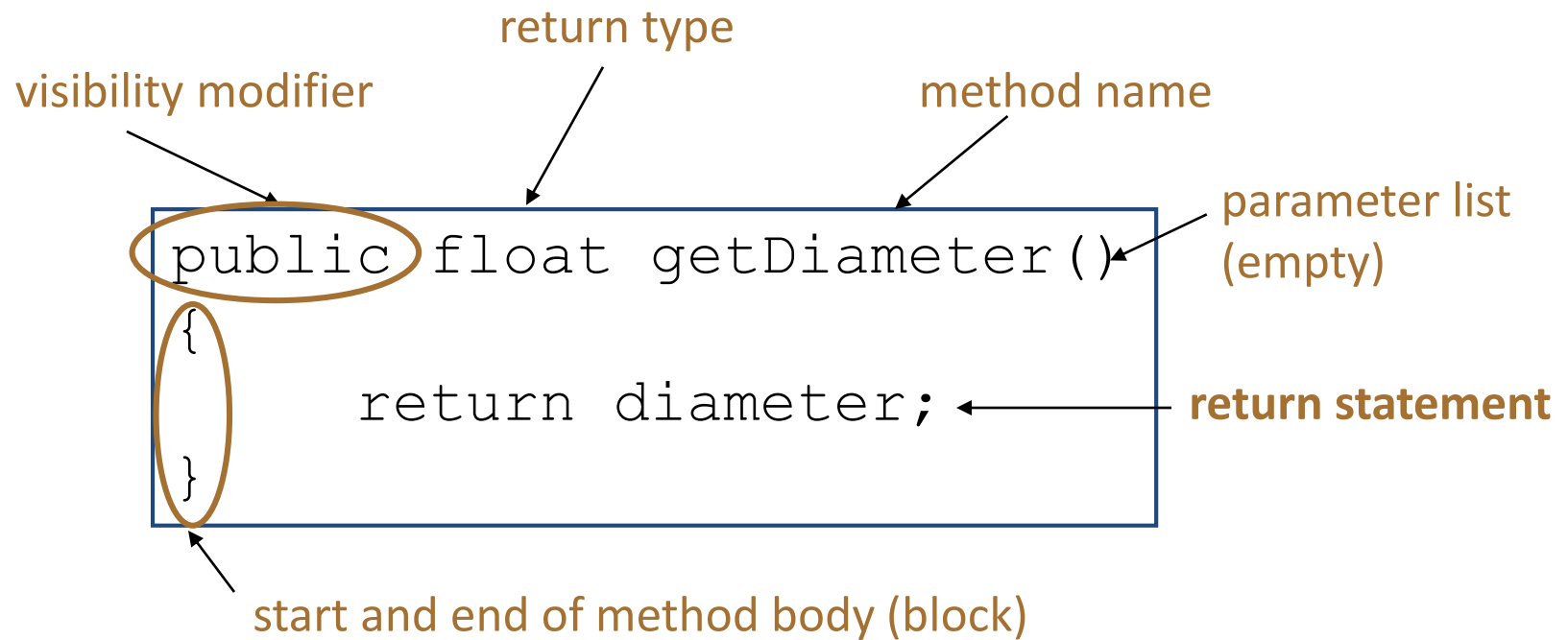
Setters and Getters to Safeguard Data



Getters (Accessor Methods)

- **Accessor** methods
 - return information about the **state** of an object
 - i.e. the values stored in the fields.
- A **'getter'** method
 - is a specific type of **accessor** method and typically:
 - **contains a return statement**
(as the last executable statement in the method).
 - defines a **return type**.
 - **does NOT change the object state**.

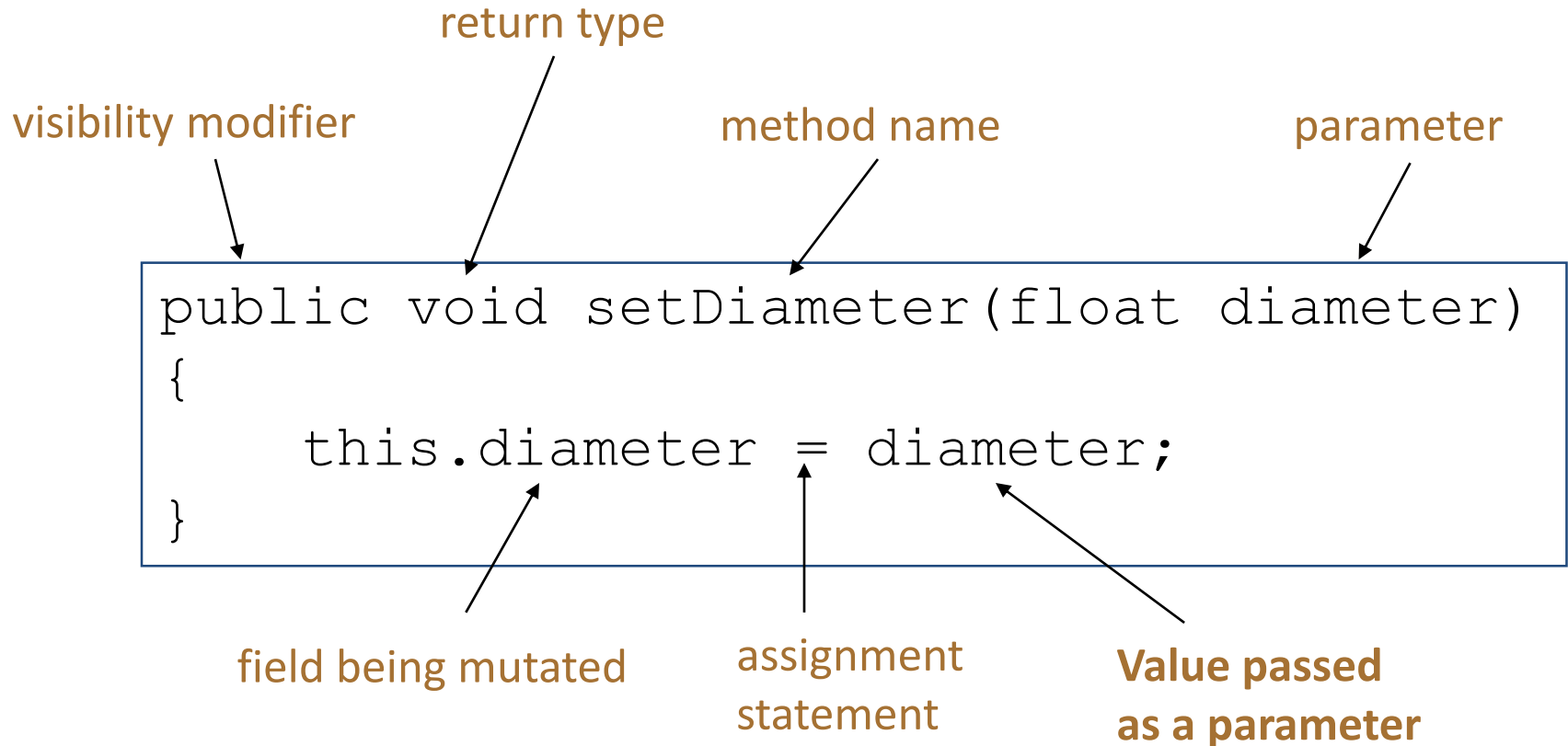
Getters



Setters (Mutator methods)

- **Mutator** methods
 - change (i.e. mutate!) an object's state.
- A **'setter'** method
 - is a specific type of **mutator** method and typically:
 - contains an **assignment statement**
 - takes in a **parameter**
 - **changes the object state.**

Setters



Getters/Setters

- For **each instance field** in a class, you are normally asked to write:
 - A **getter**
 - Return statement
 - A **setter**
 - Assignment statement

Refactoring Spot 7.0: Getters

```
public class Spot{  
    private float xCoord, yCoord;  
    private float diameter;  
    private int red, green, blue;  
  
    //constructors...  
    //display method...  
    // move method...  
    // colour methods...  
  
    public float getDiameter(){  
        return diameter;  
    }  
}
```

```
    public float getXCoord(){  
        return xCoord;  
    }  
  
    public float getYCoord(){  
        return yCoord;  
    }  
  
    public int getRed(){  
        return red;  
    }  
}
```

```
    public int getGreen(){  
        return green;  
    }  
  
    public int getBlue(){  
        return blue;  
    }  
  
    public int getGray(){  
        return gray;  
    }  
  
} //end Spot class
```

Refactoring Spot 7.0: **Setters**

```
public class Spot{  
    private float xCoord, yCoord;  
    private float diameter;  
    private int red, green, blue;  
  
    //constructors...  
    //display method...  
    // move method...  
    // colour methods...  
    // assessor methods...  
  
    public void setDiameter (float diameter){  
        this.diameter = diameter;  
    }  
}
```

```
    public void setXCoord (float xCoord){  
        this.xCoord = xCoord;  
    }
```

```
    public void setYCoord (float yCoord){  
        this.yCoord = yCoord;  
    }
```

```
    public void setRed (int red){  
        this.red = red;  
    }
```

```
    public void setGreen (int green){  
        this.green = green;  
    }
```

```
    public void setBlue (int blue){
```


Spot Class – Version 7.0

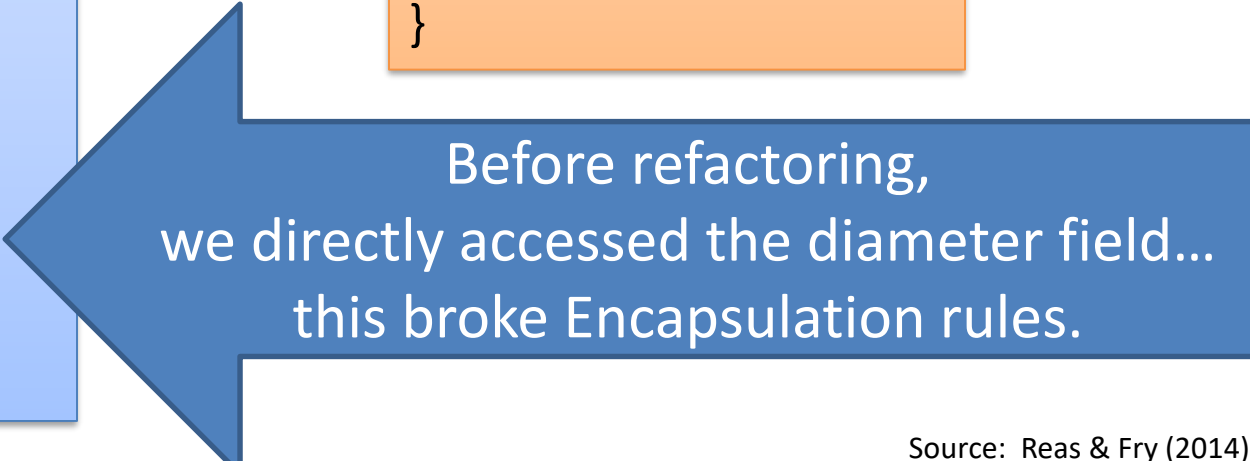
```
Spot sp;

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

void draw()
{
  background(0);
  sp.colour(255, 0, 0);
  sp.diameter = 30000;
  sp.display();
}
```

```
class Spot{
  float xCoord, yCoord;
  float diameter;
  int red, green, blue;

  // constructors...
  // display method...
  // colour methods...
  // move methods...
}
```



Before refactoring,
we directly accessed the diameter field...
this broke Encapsulation rules.

Refactoring Spot 7.0 – getters and setters

```
Spot sp;

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

void draw()
{
  background(0);
  sp.colour(255, 0, 0);
  sp.setDiameter(30000);
  sp.display();
}
```

```
class Spot{
  private float xCoord, yCoord;
  private float diameter;
  private int red, green, blue;

  // constructors...
  // display method...
  // colour methods...
  // move methods...
  //getter methods...
  //setter methods...

  public void setDiameter (float diameter) {
    this.diameter = diameter;
  }
}
```

Now we update via the appropriate setter

Review – Encapsulation steps

We have:

- Wrapped the fields & methods into a single unit
- Hidden our fields (they are **private**)
- Implemented **getter** and **setter** methods
 - to view/update the fields.

Enforced the Encapsulation rules

Encapsulation Step

1. **Wrap** the data (fields) and code acting on the data (methods) together as single unit.

2. **Hide** the fields from other classes.

3. **Access** the fields only through the methods of their current class.

But Our Design Still Smells!


BECAUSE

While we can no longer directly access the field/property,

we can still set the field values
to undesirable values by passing in a parameter ...
e.g. 30000

→ We need **validation!**

Topics list

1. Recap: Version 6.1
2. Our design smells!
3. **Encapsulation**
4. Refactoring Spot:
 - Access Modifiers
 - Accessors and Mutators
 -  – Validation

Improving the constructor

```
Spot(float xCoord, float yCoord, float diameter)
{
    this.xCoord = xCoord;
    this.yCoord = yCoord;
    this.diameter = diameter;
}
```

Current constructor
with no validation.

Improving the constructor

```
Spot(float xCoord, float yCoord, float diameter)
{
    this.xCoord = xCoord;
    this.yCoord = yCoord;
    if ((diameter >= 20) && (diameter <= 50)) {
        this.diameter = diameter;
    }
    else{
        this.diameter = 20;
    }
}
```

Updated constructor
with some validation.

Note: in the constructor, you typically set the field to a **default** value if invalid data was entered.

Improving the **setter** / mutator

```
public void setDiameter (float diameter) {  
    if ((diameter >= 20) && (diameter <= 50)) {  
        this.diameter = diameter;  
    }  
}
```

Note: The validation done at constructor level must be **repeated at setter level** for that field
→ **data integrity!**

However, in setter methods, you typically do not update the field's value if invalid data was entered (no “else” branch).

Summary - Encapsulation (data hiding)

- Hide fields
 - Access them only through methods of the class
e.g. **getters** & **setters**
- Make the
 - **class public**
 - and the **fields private**
- 4 Access Levels
(2 for processing)
 - package
 - **private**
 - **public**
 - Protected
- Accessors
 - **get**
- Mutators
 - **set**
- Write a getter & setter for each each field
- **Validation**
 - Test min / max values
 - in **constructor**
 - Apply default if value fails
 - in **setter**
 - Ignore the update if value fails

Summary continued

Encapsulation – Steps

1. Wrap Fields & Methods in single file
2. Hide the fields from other classes using private
3. Access only through getter & setters
4. Apply validation in constructors & setters

Questions?



References

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