CIS 351-Data Structure-Class and Objects
Jan 30, 2020

Dr. Farzana Rahman

Syracuse University

Cohesion and Coupling

Coupling

- The degree to which two classes (or methods) are "tied together." How likely is it that changing the internals of one will require modification of the other
- We aim for LOW coupling

Cohesion

- The degree to which the components of a class "hang together"
- We aim for HIGH cohesion

Problems Created by Bad Cohesion

- Hard to understand the class
- If two abstractions grouped into one class, that implies a one-to-one relationship
  - What if this changes?
- Often we specialize a class along a dimension
  - This new thing is like the existing one except we extend it in one area (dimension)
  - Problems arise when each of the several abstractions need such specialization

The "Multiplicity" Problem

- Consider an Account class that holds:
  - Customer name, address, tax ID, Account status, etc.
- What if one customer needs two accounts?
  - Two Account objects, but each stores name and address
- What if one account has two owners?
  - You can't do this, unless you create a collection in each Account to hold owner info

How to Achieve Better Cohesion

- Some of this is just good OO experience
  - Eliminate redundancy
  - Attributes should have a single value and should not have structure (repeating groups of things)
  - Attributes always describe an instance of its containing class
  - That's what attributes are all about. State values that define a particular instance
- Note: there are always tradeoffs! Sometimes we combine abstractions into one class for efficiency.

Some Design Proposals

In this design, SeatingChart has an array of Student Objects.

Let's assume there is a GUI class displaying the seating chart of the students.

Student
-name: String -row: int -column: int -friends: ArrayList<Student>
+Student(name: String, row: int, column: int) +Student(name: String) +getName(): String +getRow(): int +getColumn(): int +setRow(row: int) +setColumn(column: int) +addFriend(friend: Student) +getUnhappiness(): double +toString(): String

SeatingChart
-seats: Student[][]
+SeatingChart(rows: int, columns: int) +SeatingChart(fileName: String) {exceptions=FileNotFoundException, FileFormatException} +SeatingChart(file: File) {exceptions=FileNotFoundException, FileFormatException} +getRows(): int +getColumns(): int +getStudent(row: int, column: int): Student +getStudent(name: String): Student +placeStudent(row: int, column: int, student: Student) +getTotalUnhappiness(): double +swap(row1: int, column1: int, row2: int, column2: int) +stepGreedy() +solveGreedy() +toString(): String +toStringVerbose(): String +save(fileName: String) {exceptions=FileNotFoundException} +save(file: File) {exceptions=FileNotFoundException} +export(fileName: String) {exceptions=FileNotFoundException} +export(file: File) {exceptions=FileNotFoundException}

Design Proposal

- Let's combine the GUI code and the SeatingChart class. Each call to step or solve can automatically refresh the display!

  - Reduces Cohesion
  - Makes it harder to test and reuse the SeatingChart logic.
  - Makes it harder to modify the GUI.

Some Design Proposal

- Let's give Student objects a reference to the SeatingChart they are in, and give them responsibility for moving themselves:

```
bob.moveTo(3, 5)
```

  - logic is similar to placeStudent
  - no longer need setRow + setColumn methods?
- Not actually a bad idea, but it does increase coupling. In this design the Student class needs to "know about" the SeatingChart class.

Some Design Proposal

- Let's add code to the Student class for abbreviating names:
  - getName() → “Nathan Sprague”
  - getAbbreviatedName → “N. Sprague” Splits the string, pulls out the first letter etc.
- (Users have requested this as an option)
- Lowers cohesion. Perhaps create a Name class, or a Utility class that can perform these conversions.
- Otherwise, introducing a Teacher class will require us to copy-paste this functionality.

Objects, Classes and Constructors

Class Circle

```
public class Circle
{

  public double x, y;     // centre of the circle
  public double r;        // radius of circle

  //Methods to return circumference and area
  public double circumference()
  {
     return 2*3.14*r;
  }
  public double area()
  {
     return 3.14 * r * r;
  }
}
```

Visibility Modifiers

By default, the class, variable, or data can be accessed by any class in the same package.

- **`public`**

  The class, data, or method is visible to any class in any package.

- **`private`**

  The data or methods can be accessed only by the declaring class.

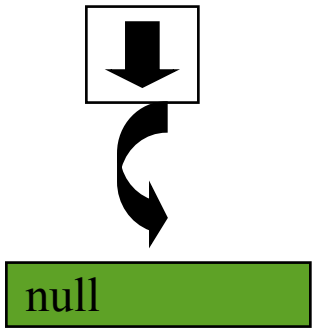
- The get and set methods are used to read and modify private properties.

Class of Circle

```
Circle aCircle;
Circle bCircle;
```

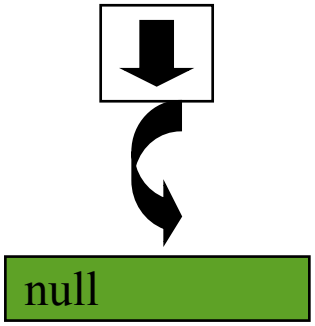
- aCircle, bCircle simply refers to a null, not an actual object.

aCircle



Points to nothing (Null Reference)

bCircle

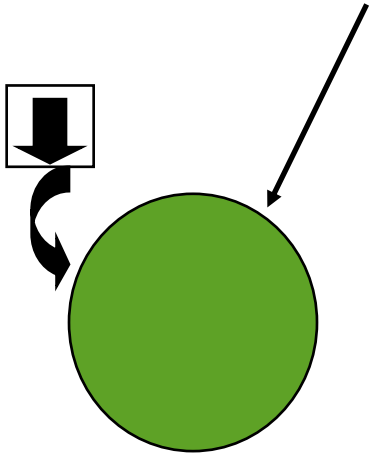


Points to nothing (Null Reference)

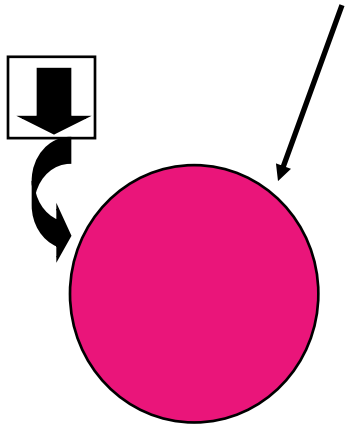
Creating objects of a class

- Objects are created dynamically using the *new* keyword.

```
aCircle = new Circle()
```



```
bCircle = new Circle()
```



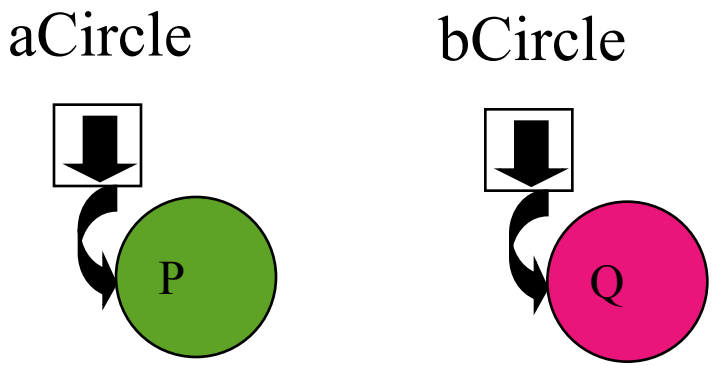
Now aCircle and bCircle refer to Circle objects

Assigning one object to the other

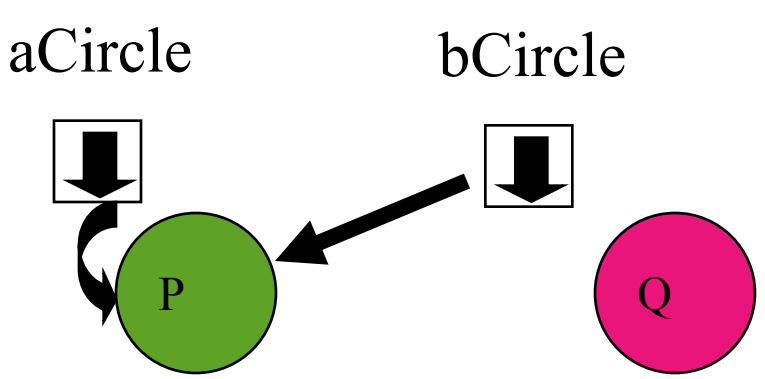
```
aCircle = new Circle();
bCircle = new Circle() ;
```

bCircle = aCircle;

Before Assignment



Before Assignment



Constructors

```
Circle(double r) {
  radius = r;
}

Circle() {
  radius = 1.0;
}

myCircle = new Circle(5.0);
```

Constructors are a special kind of methods that are invoked to construct objects.

Constructors, cont.

•A constructor with no parameters is referred to as a *default constructor*.

•Constructors must have the same name as the class itself.

•Constructors do not have a return type — not even void.

•Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Instance Variables, and Methods

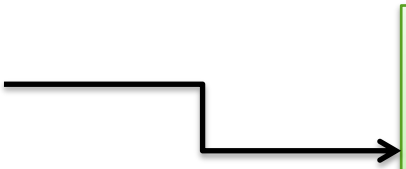
- Instance variables belong to a specific instance of a class (i.e. any object)
- Instance methods are invoked by an instance of the class (i.e. any object)

Class Variables, Constants, and Methods

- Class variables are shared by all the instances of the class.
- Class methods are not tied to a specific object.
- Class constants are final variables shared by all the instances of the class.

Constructor calling

- Point p1 = new Point () ;



```
public Point()
{ x = 0 ;
  y = 0 ;  }
```

- Point p2 = new Point(2, 3);

```
public Point(int x_val, int y_val)
{   setX (x_val);
    getY (y_val);
}
```

- Point p1 = new Point () ;

  p1.setX(2);

  p1.setY(3);

Scope of Variables

- The scope of instance and class variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

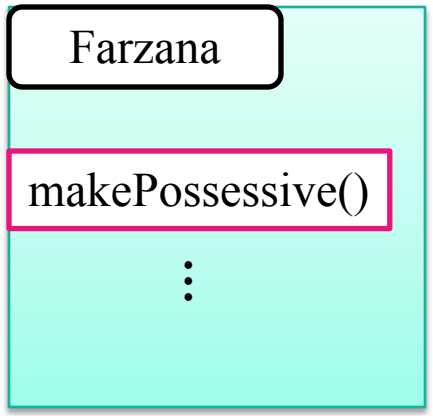
```
public class Word
{
  private String  inword;
  public Word(String str)
  {
     inword = str;
  }
  public String makePossessive()
  {
     return inword+"'s" ;
  }
  …
  …
  …
}
```

Different objects will have different instance variables

Different objects will execute their own method

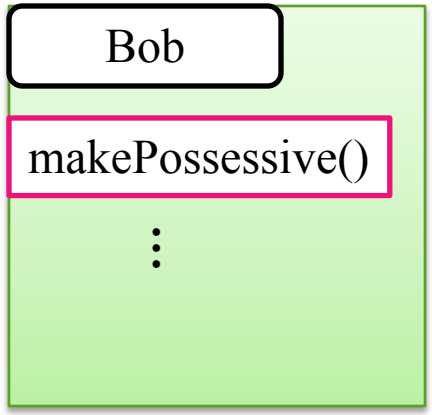
Instance variable tied to name object

```
Word name = new Word("Farzana");
```



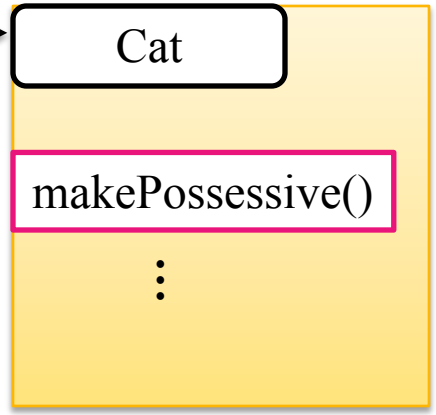
Instance variable tied to fname object

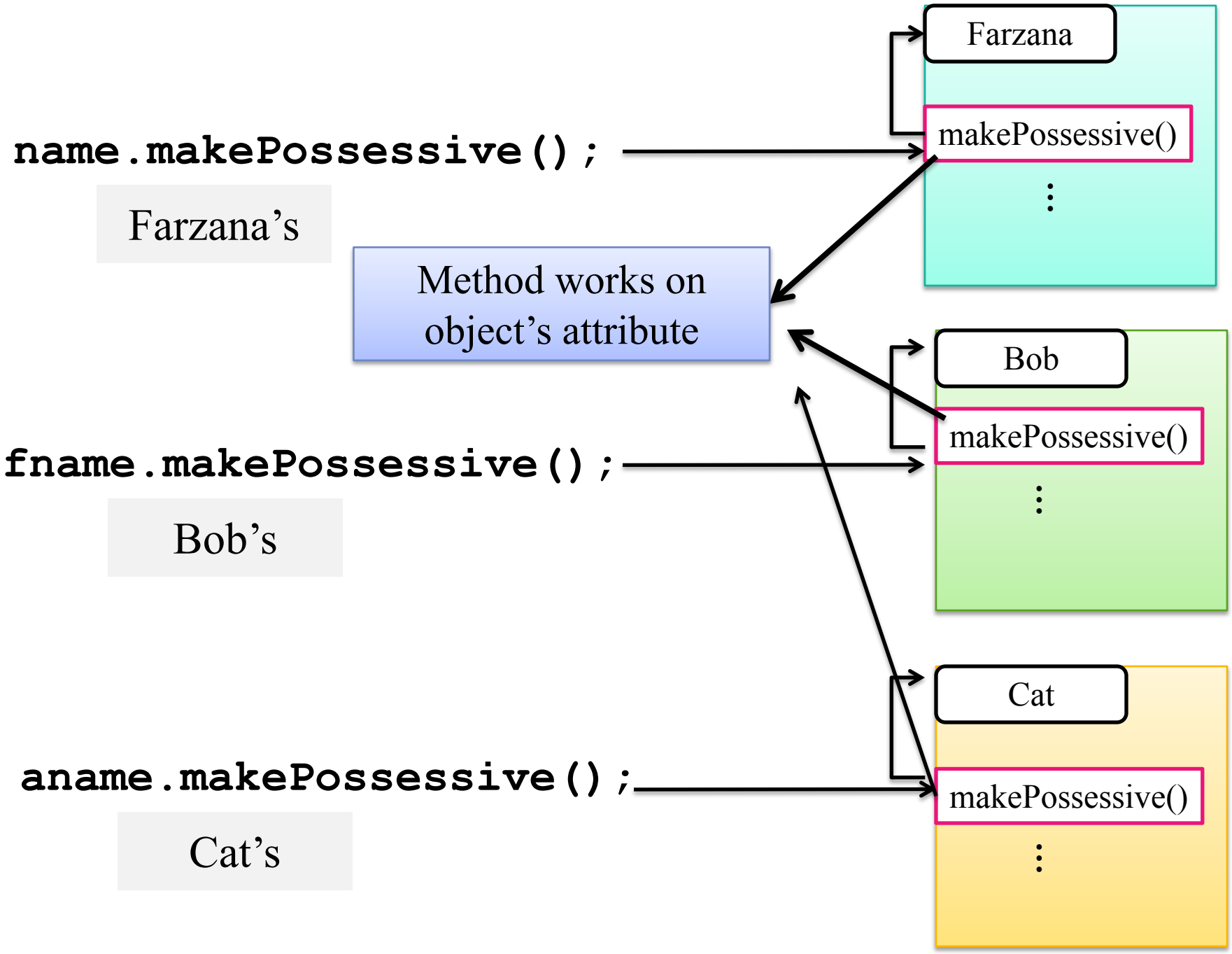
```
Word fname = new Word("Bob");
```



Instance variable tied to aname object

```
Word aname = new Word("Cat");
```




name.makePossessive();
Farzana's
Method works on object's attribute
fname.makePossessive();
Bob's
aname.makePossessive();
Cat's
Farzana
makePossessive()
Bob
makePossessive()
Cat
makePossessive()

Accessor (setter) and Mutator (getter) Methods

```
public class Point ()
{
   private int x, y;

   public Point(){
     x = 0 ;
     y = 0 ; }

  public Point(int x_val, int y_val){
     setX (x_val);
     getY (y_val); }

  public void setX(int X)
     { x = X ; }

  public void setY(int Y)
     { y = Y ; }

  public int getX()
     { return x ; }

  public int getY()
     { return y ; } }
```

Setter and Getter methods are used to set and retrieve the values of instance variables

Initializing objects

- Currently it takes 3 lines to create a `Point` and initialize it:

```
Point p = new Point();
p.x = 3;
p.y = 8;                              // tedious
```

- We'd rather pass the fields' initial values as parameters:

```
Point p = new Point(3, 8);    // better!
```

  - We are able to this with most types of objects in Java.

Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.

- Write a constructor for Point objects that accepts no parameters and initializes the point to the origin, (0, 0).

```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```