(remedial) Java

anastasia.bezerianos@lri.fr

Object Oriented Programming

Object Oriented Programming

- All languages make abstractions
- OOP attempts to represent the real world as close as possible, i.e. closer to the problem space
- Use of the notion of **objects**




Object Oriented Programming

- Represent the real world

Car 

Object Oriented Programming

- Represent the real world

Car
Make
Color
Horsepower
MPG
Manual
Kilometers

Object Oriented Programming

- Objects group together characteristics:
  - Primitives (int, double, char, etc..)
  - Objects (String, etc...)

Car

```
String Make
String Color
int Horsepower
double MPG
boolean Manual
int Kilometers
```

Why use classes?

- We could use just primitives …

```
// my car
String myCarMake;
String myCarColor;
int myCarHorsepower;
double myCarMPG;
boolean myCarManual;
int myCarKilometers;
```


Why use classes?

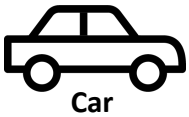
- We could use just primitives …

```
// my car
String myCarMake;
String myCarColor;
int myCarHorsepower;
double myCarMPG;
boolean myCarManual;
int myCarKilometers;

// my father's car
String dadsCarMake;
String dadsCarColor;
int dadsCarHorsepower;
double dadsCarMPG;
boolean dadsCarManual;
int dadsCarKilometers;
```

- What if I had 800 cars …

Why use classes?



Car

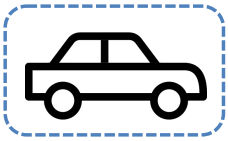
Make
Color
Horsepower
MPG
Manual
Kilometers

Make = …
Color = …
Horsepower = …
MPG = …
Manual = …
Kilometers = …

myCar

Why use classes?

Car



Make
Color
Horsepower
MPG
Manual
Kilometers

myCar

Make
Color
Horsepower
MPG
Manual
Kilometers

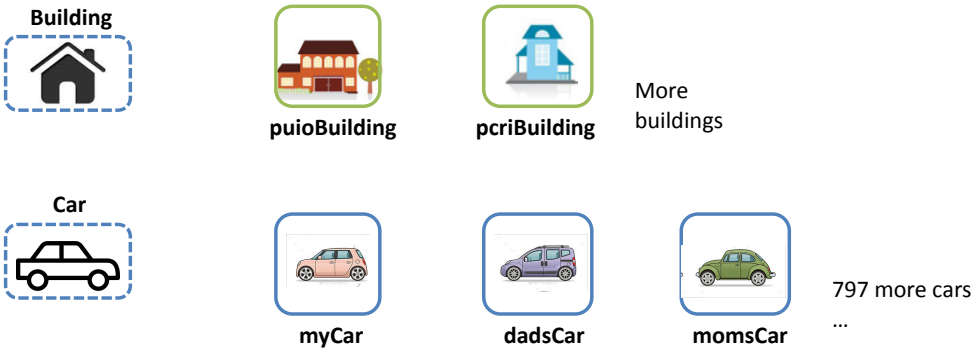
dadsCar

Make
Color
Horsepower
MPG
Manual
Kilometers

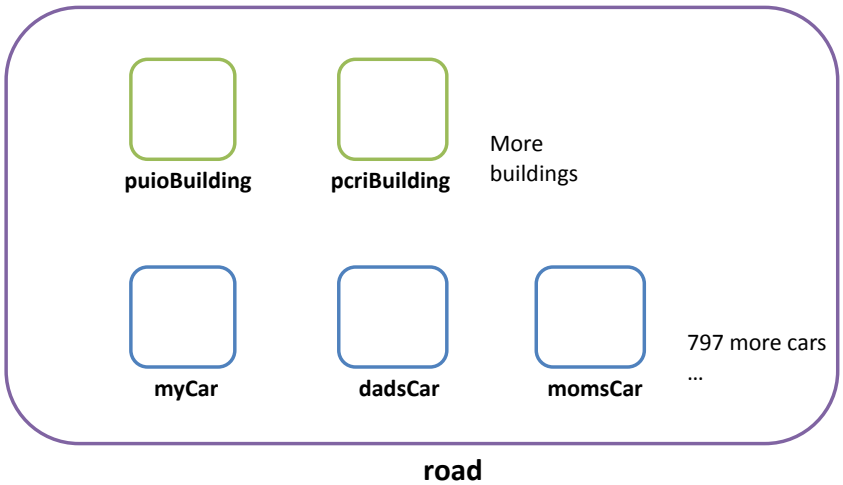
momsCar

797 more cars …

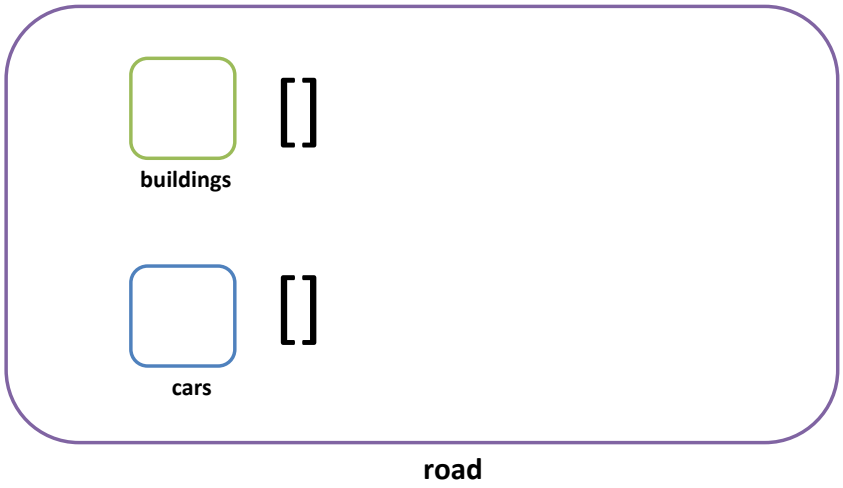
Why use classes?



Why use classes?



Why use classes?



Object and Class (1)

- Everything is an **object.**
- A program is a collection of objects that tell each other what to do (pass messages)
- Similar **objects** are grouped together in an object **class**.
  - A class is a "pattern template" for creating similar objects.

Object and Class (2)

- With the keyword **class** we can define abstract data types, i.e. classes (fundamental concept of OOP).
- We use these abstract data types in the same way as we use the predefined Java data types.
  - Can create variables of the specific type (class), and give them concrete values (object/instance)
  - The instances (objects) of a class share the same characteristics (e.g., all cars have a make, model, color, etc.).
  - But each object has its own state, so my car is a pink peugeot 104, my mother's car is a green volkswagen beetle, etc.

Defining classes

Let's declare a car class

```
public class Car {



}
```


Let's declare a car class

```
public class Car {
```

fields: data/characteristics

methods: data manipulation

msg treatment

```
}
```

Class fields

```
public class Car {


    TYPE varname;
    TYPE varname = some_value;









}
```


Class fields

```
public class Car {

    String make;
    String color;
    int horsepower;
    double mpg;
    boolean manual;
    int kilometers = 0;









}
```

Class fields

```
public class Car {

    String make;
    String color;
    int horsepower;
    double mpg;
    boolean manual;
    int kilometers = 0;
    XXXX YYYY;



}
```


Class fields

```
public class Car {

    String make;
    String color;
    int horsepower;
    double mpg;
    boolean manual;
    int kilometers = 0;
    Car[] overtaken;



}
```

Class methods

```
public class Car {

   String make;
   String color;
   int horsepower;
   double mpg;
   boolean manual;
   int kilometers = 0;
   Car[] overtaken;

   void move (int speed, int time) {
      kilometers += speed*time;
      System.out.println("Covered " + kilometers + "km");
   }
}
```


Class overview

```
public class Car {

   String make;
   String color;
   int horsepower;
   double mpg;
   boolean manual;
   int kilometers = 0;
   Car[] overtaken;

   void move (int speed, int time) {
      kilometers += speed*time;
      System.out.println("Covered " + kilometers + "km");
   }
}
```

Class
Definition

Note on Class declaration

In java:

- Class names are Capitalized
- 1 Class = 1 file
- a Class with a `main` function can be run

Class overview

Class

Instance

(a concrete object of that class)

```
Car myCar = new Car();
```

But what make, color, …?

Class Constructor

```
public class CLASSNAME {

   CLASSNAME ( ) {
   }

   CLASSNAME ([ARGUMENTS]) {
   }
}


CLASSNAME obj1 = new CLASSNAME();
CLASSNAME obj2 = new CLASSNAME([ARGUMENTS]);
```


Car constructors

```
public class Car {
   String _make;
   String _color;
   int _horsepower;
   double _mpg;
   boolean _manual;
   int _kilometers = 0;

   public Car(String mymake, String mycolor){
      _make = mymake;   // or this._name = myname;
      _color = mycolor; // or this._color = mycolor;
   }

   void move (int speed, int time) {
      _kilometers += speed*time;
      System.out.println("Covered " + _kilometers + "km");
   }
}
```

Class Constructor characteristics (1)

- A constructor is a particular method
- It is used to initialize an object, it initializes fields, and guarantees invariants at the objects' creation
- It is invoked with the keyword **new** once per object, and it reserves memory for the newly created object

Class Constructor characteristics (2)

- Constructor name = Class name
- No return type, never returns anything
- All classes need at least one constructor, if none defined than Java adds a default one

```
CLASSNAME ( ) {
}
```

- If a constructor is defined, the Java compiler no longer creates a default constructor

Inter-constructor calls

- Can call one constructor method from another using the keyword **this** (more later). This process is called constructor overloading.
- The call to the constructor must be the first call

```
public Car(){
    this("nomake", "nocolor"); // I want a default value
}

public Car(String myname, String mycolor){
    _name = myname;  // or this.name = myname;
    _color = mycolor;   // or this.color = mycolor;
}
```


Using classes

Classes and Instances

```
// class Definition (pattern)
public class Car {
    …
}

// class Instances (concrete objects)
Car myCar = new Car ("peugeot", "pink" );
Car momsCar = new Car ("volkswagen", "green" );
```


Classes and Instances

```
// class Definition
public class Car {
    …
}

// class Instances
Car myCar = new Car ("peugeot", "pink" );
Car momsCar = new Car ("volkswagen", "green" );
```

- Each instance (myCar, momsCar) has their own values for their fields (e.g., different make or color).

Accessing members of a class

- Members of a class (fields, methods) can be accessed with the notation **“.”** applied after the reference to an object

Accessing members of a class

- Members of a class (fields, methods) can be accessed with the notation **“.”** applied after the reference to an object
- For fields `Object.Fieldname`

```
Car myCar = new Car ("peugeot”, “pink” );

System.out.println(myCar._make); // returns peugeot
```

Accessing members of a class

- Members of a class (fields, methods) can be accessed with the notation **“.”** applied after the reference to an object
- For methods `Object.Methodname([ARGUMENTS ])`

```
Car myCar = new Car ("peugeot", “pink” );

myCar.move(85,30); // updates the variable kilometers
```


Accessing members of a class

- With respect to classic functions (e.g. in C), we chose one of the objects to act on:
  - C: equals(o1,o2)
  - Java: o1.equals(o2)

```
Class Car{
   boolean equals (Car m){
   ... }
}
```

```
class Main{
   public void todo{
      Car c1 = ...
      Car c2 = ...
      if ( c1.equals(c2)){
      ...}
   }
}
```

this

- **this** (or self, it) corresponds to a reference to the object on which a method is called
- Clarifies the scope of the variable (if same name)

```
Class Car{
   boolean equals (Car c){
      if (this == c)
         return true;
   }
}
```

```
class Main{
   public void todo{
      Car c1 = ...
      Car c2 = ...
      if ( c1.equals(c2)){
      ...}
   }
}
```

Here **this** takes the value of c1 and **c** of c2

this example

```
public class Car {
   int kilometers = 0;

   Car(){
      move(30);
      System.out.println("Covered " + kilometers + "km");
   }
   void move (int kilometers) {
      kilometers += kilometers;
      System.out.println("Covered " + kilometers + "km");
   }

   public static void main (String args[]){
      Car car1 = new Car();
   }
}
```

- Only method-level kilometers is updated

this example

```
public class Car {
   int kilometers = 0;

   Car(){
      move(30);
      System.out.println("Covered " + kilometers + "km");
   }
   void move (int kilometers) {
      this.kilometers += kilometers;
      System.out.println("Covered " + kilometers + "km");
   }

   public static void main (String args[]){
      Car car1 = new Car();
   }
}
```

- Object-level kilometers is updated

Exercise 1 - part 1

www.lri.fr/~anab/teaching/remedialJava/ex-session2.pdf

Hint: use Book[] inside Library

Reminder:

- Class definition

```
public class CLASSNAME {
   TYPE varname;                    // field
   TYPE varname = some_value;       // initialized field
   CLASSNAME ([ARGUMENTS]) { … } // constructor method
   TYPE method_name ([ARGUMENTS]) {…} // another method
}
```

- Class use

```
CLASSNAME obj = new CLASSNAME([ARGUMENTS]);
obj.method_name([ARGUMENTS]);
```

References vs Values

Primitives vs References

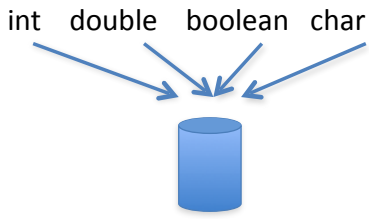
- Primitives are basic Java types
  - int, char, double, float, long, short, boolean, byte
  - The actual value is stored in the variable (memory)
  - These are not objects (almost everything else is)

Primitives vs References

- Primitives are basic Java types
  - int, char, double, float, long, short, boolean, byte
  - The actual value is stored in the variable (memory)
  - These are not objects (almost everything else is)
- Reference types are arrays and objects
  - String, int[], Car, …

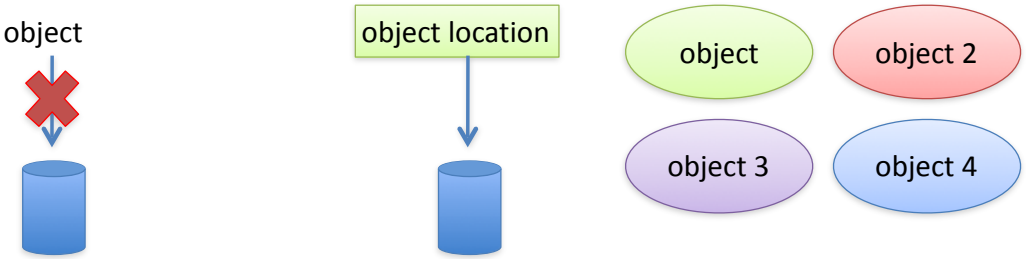
How Primitives are stored

- Variables are like fixed size memory buckets
- Primitives are small enough to fit in them



How Objects are stored

- Objects are too big to fit in a variable
- Instead
  - They are stored somewhere else
  - Variables store a number that locates the object



References

- The object's location (address) is called a reference
- The operator == compares references

```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
```

- Does car1 == car2?

NO

References

- The object's location (address) is called a reference
- The operator == compares references

```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
```

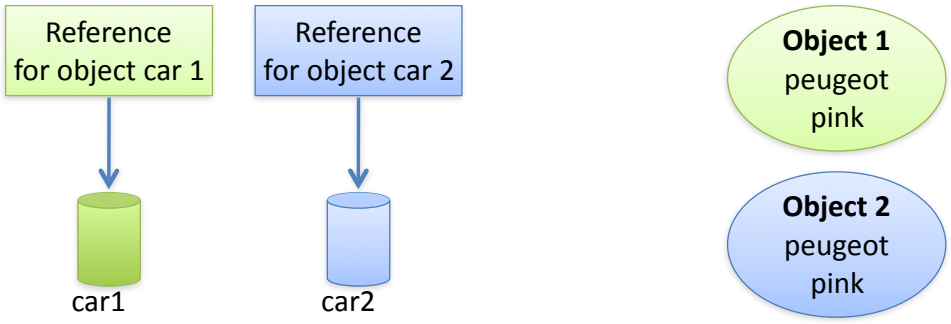
- Does car1 == car2?

NO

This is why we use `string1.equals(string2)`

References

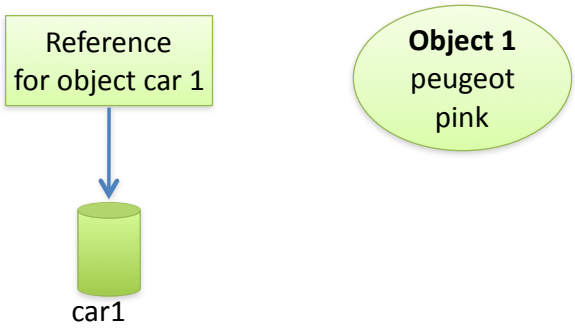
```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
```



References

- Using = on a field updates its *value* on the object

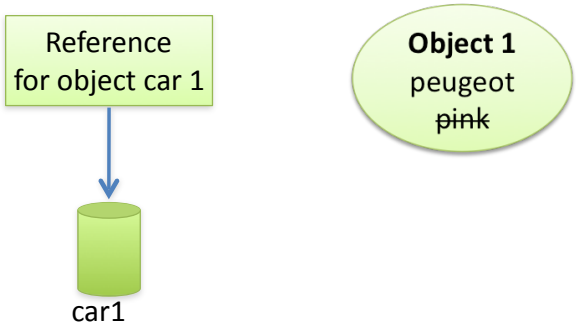
```
Car car1 = new Car ("peugeot", "pink");
car1.color = "red";
```



References

- Using = on a field updates its *value* on the object

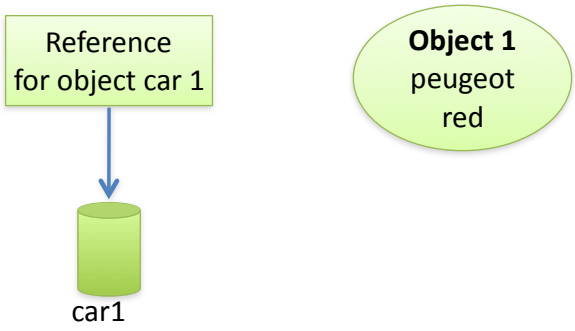
```
Car car1 = new Car ("peugeot", "pink");
car1.color = "red";
```



References

- Using = on a field updates its *value* on the object

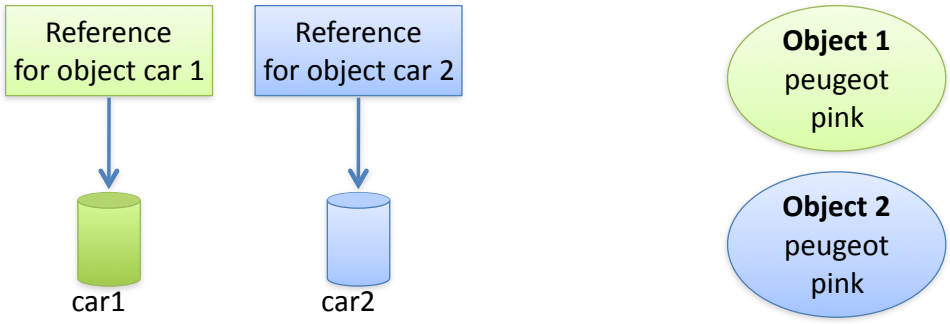
```
Car car1 = new Car ("peugeot", "pink");
car1.color = "red";
```



References

- Using = on an object variable updates the *reference* to the object

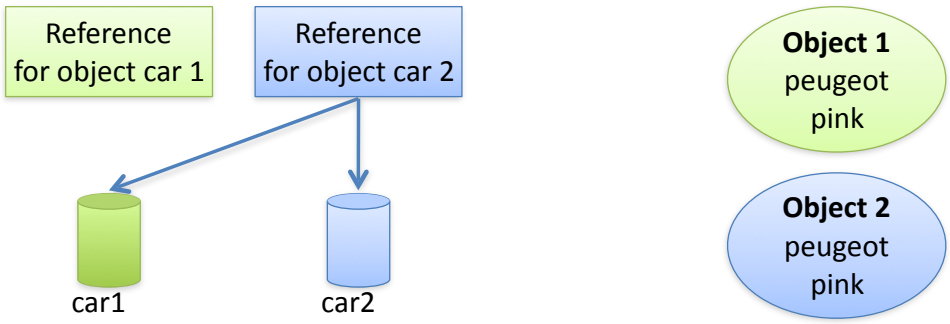
```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
car1= car2;
```



References

- Using = on an object variable updates the *reference* to the object

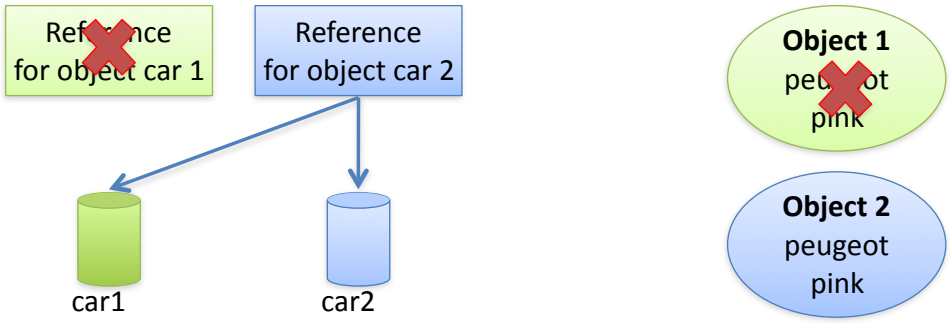
```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
car1= car2;
```



References

- Using = on an object variable updates the *reference* to the object

```
Car car1 = new Car ("peugeot", "pink");
Car car2 = new Car ("peugeot", "pink");
car1= car2;
```



References

- Using [] or •
  - Follows the reference to the object
  - May modify the object, but never the reference

Imagine following directions to a house, you can modify the furniture inside (analogous to following the reference of an object and changing its fields)

Methods and References

```
void dofoo (int x, int[] many_ints, Car c){
   x = 100;
   many_ints[0] = 100;
   c.make = "100";
}

…

int i = 0;
int[] j = {0};
Car k = new Car ("peugeot", "pink");
dofoo(i,j,k);
```

What are the values of i,j,k?

Methods and References

- Primitives (int, double, boolean)
  - The actual value is stored in the variable (memory)
  - So the scope of the variable affects value

  (pass by value)

- Reference types (arrays and objects)
  - The address (ref) to the object is stored in the variable
  - So even if the variable scope changes, any changes to the object remain (as the variable points to the object)

  (pass by reference)

Type and Class

A side note:

- Variables have a Type
- Basic types have a value
- References have a Class

```
public static void main (String args[]){
   int x  = 0; // type is int, value is 0

   String s = args[0];
   Car k = new Car ("peugeot", "pink");
}
```

Type Class

Exercise 1 - part 2 & 3

www.lri.fr/~anab/teaching/remedialJava/ex-session2.pdf

Hint: use Book[] inside Library

Reminder:

- Class definition

```
public class CLASSNAME {
   TYPE varname;                  // field
   TYPE varname = some_value;     // initialized field
   CLASSNAME ([ARGUMENTS]) { … } // constructor method
   TYPE method_name ([ARGUMENTS]) {…} // another method
}
```

- Class use

```
CLASSNAME obj = new CLASSNAME([ARGUMENTS]);
obj.method_name([ARGUMENTS]);
```

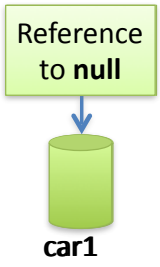

Null Reference

A side note: null

- As you continue with Java you may come across the keyword **null**, this is the value of a reference that is not pointing anywhere

```
Car car1; // variable car1 stores a ref
          // that points nowhere (null)

car1 = new Car ("peugeot", "pink");
// now it points to a new object
```



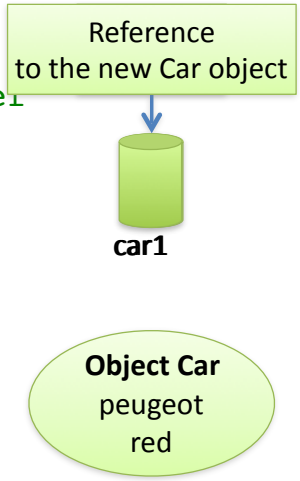
Null Reference

A side note: null

- As you continue with Java you may come across the keyword **null**, this is the value of a reference that is not pointing anywhere

```
Car car1; // variable car1 stores a ref
         // that points nowhere (null)

car1 = new Car ("peugeot", "pink");
// now it points to a new object
```



Null Reference

A side note: null

- As you continue with Java you may come across the keyword **null**, this is the value of a reference that is not pointing anywhere

```
Car car1; // variable car1 stores a ref
         // that points nowhere (null)

car1 = new Car ("peugeot", "pink");
// now it points to a new object
```

You could test if a reference is null (e.g., `if(car1==null)`, but this is a last resort practice, there are usually ways to write your code so this is not needed

Static types and methods

static

- The keyword static can be applied to fields and methods
- It states that the field or method:
  - Is defined at the class declaration (not the object)
  - Is not unique for each instance (object), but is shared between all objects of the class

static

```
public class Car {
   static int noOfCarsMade = 0;
}

Car car1 = new Car ();
Car car2 = new Car ();
Car.noOfCarsMade = 2;
```

- What is `car1.noOfCarsMade`? `car2.noOfCarsMade`?

static example

- Keep track of cars made

```
public class Car {
   int noOfCarsMade = 0;

   Car() {
      ++noOfCarsMade;
   }
}
```

- What did I forget?

static **field** example

- Keep track of cars made

```
public class Car {
   static int noOfCarsMade = 0;

   Car() {
      ++noOfCarsMade;
   }
}
```


static method example

```
public class Car {
   static void carColor( Car mycar){
      System.out.println(mycar.color); // or this.color
   }
}
```

- Or …

```
public class Car {
   void carColor(){
      System.out.println(color);
   }
}
```

- Depending on whether the call is made by a static method

static method

- Non-static methods can reference static methods, but not the other way round, why?

```
public class Car {
   String numberPlate;
   static void whoami(){
      System.out.println(numberPlate);
   }
}
```


main

- Why is main static?

```
public static void main (String[] args) {

}
```

main

- Why is main static?

```
class Car {
   void move (int x){
      _kilometers += x;
   }

   public static void main (String[] args) {
      // I cannot call move(100);
      // as it needs to be static,
      // BUT I can run it through an object

      Car c = new Car();
      c.move(100);
   }
}
```


Principles of OOP

Principles of OOP

- Abstraction (classes …)
  - Grouping functionality related to original objects/ concepts we are trying to represent (hiding the implementation details, related to encapsulation …)
- One object, one responsibility
- Encapsulation
- Inheritance (later)
- Polymorphism (later)

One object, one responsibility

- If we want to reuse (types of) objects, they should not hold too much functionality
- 2 object (types) => 2 responsibilities
- We use delegation/msg passing between objects (i.e. the fact that objects can communicate) to divide responsibilities

One object, one responsibility (example)

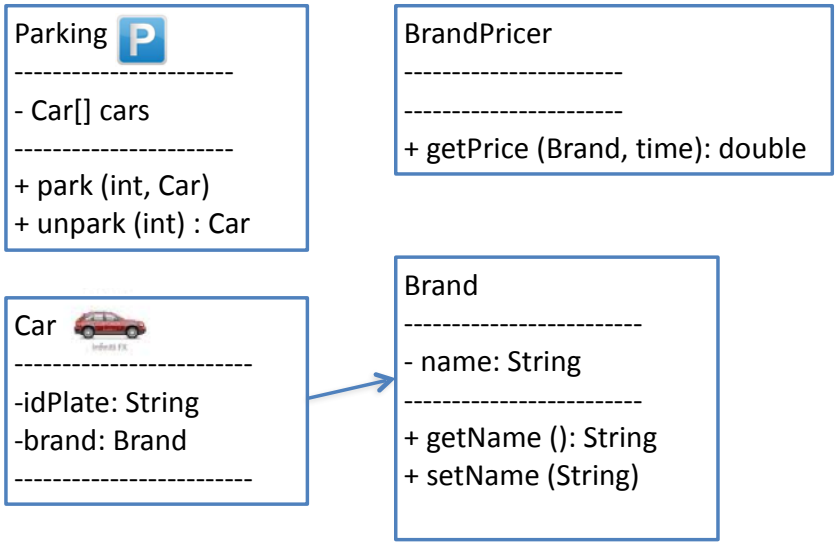
- We want to create an application that manages a garage, and calculates how much money the garage would make if all cars left the garage at once (i.e. expected gain)
- The price depends on the parking time and on the car brand
- Responsibilities?

One object, one responsibility (example)

- 4 possible responsibilities (so 4 objects)
  - Garage: store cars
  - Car: the car itself ☺
  - Brand: the car brand that affects price
  - BrandPricing: finding the price based on the brand, time and a model

One object, one responsibility (example)

- A class diagram

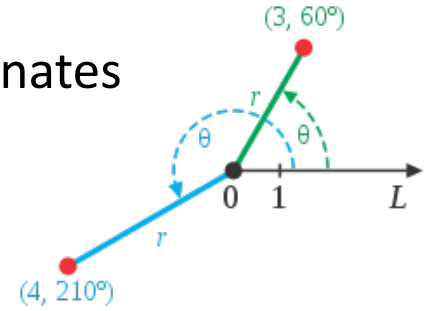


Encapsulation

- The principle that guarantees that no field in an object can be modified (and possibly be corrupted) without verification
- The state of a field can only be changed (if it is modifiable), by methods that verify the changes

Encapsulation

- An example of polar coordinates




```
class Point{
  double rho;
  double theta;

  void init (double x, double y) {
    rho = Math.hypot(x,y);
    theta = Math.atan2(x,y);
  }
}
```

```
class oneClass{
    void oneMethod (Point p){
        p.init(2.0,3.0);

        p.rho = -1; // oh, no!!!
    }
}
```

- Class Point does not guarantee encapsulation

Visibility

- Some languages (like Smaltalk) do not allow access to fields from outside the object
- Others, like C++, Java, requires us to determine the visibility of fields using visibility modifiers
- Visibility can ensure encapsulation, by restricting the direct access to fields of an object

Modifying visibility

- There are 4 visibility modifiers:
  - A **public** member is visible by everyone
  - A **protected** member is visible by inherited classes (later) and classes inside the same package
  - A member **without modifier** (or friend) is visible by classes inside the same package
  - A **private** member, is only visible inside a class

The visibility order is:

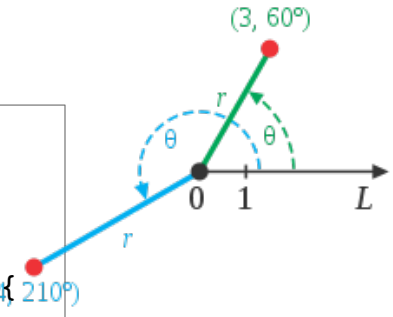
private < protected < public

Encapsulation

- Try to keep fields private

```
class Point{
  private double rho;
  private double theta;

  public void init (double x, double y){
    rho = Math.hypot(x,y);
    theta = Math.atan2(x,y);
  }
}
```




```
class oneClass{
    void oneMethod (Point p){
        p.init(2.0,3.0);
        System.out.println(p.theta);
        p.rho = -1; // last 2 lines
                    // do not compile
    }
}
```

Why modify visibility

- Controlling access to class *fields* can:
  - Protect private class information (sorta) and avoid errors in state
  - Clarifies how others can use your class
  - Keeps implementation separate from class interface (i.e. view to the world)

Visibility and maintenance

- Some advice:
  - Avoid *fields* that are public or protected (apart from constants)
  - Make *methods* public only when necessary
  - A protected method allows us to implement a common service for subclasses (later), if the package visibility is not available

Encapsulation, a problem

- Imagine we want rho to never be less than 0
  - Possibility 1: control access to rho

```
class Point{
  private double rho;
  private double theta;

  public void init (double x, double y) {
    …
  }

  public void setRho (double rho){
    // do some tests
  }
}
```


Encapsulation, a problem

- Imagine we want rho to never be less than 0
  - Possibility 2: what about initialization?

```
class Point{
  private double rho;
  private double theta;

  public void init (double x, double y) {
    …
  }

  public void setRho (double rho){
    // do some tests
  }

  public Point (double rho, double theta){
       setRho(rho);
  }
}
```

Encapsulation, a problem

- Imagine we want rho to never be less than 0
  - Possibility 2b (advanced): initialization (constructor)

```
class Point{
  private final double rho;
  private final double theta;

  public Point (double rho, double theta){
    if (rho <= 0) {
        throw new IllegalArgumentException ("Illegal rho" + rho);
    }
  this.rho = rho;       // else will not compile
  this.theta = theta;   // else will not compile
}
```

  - Can make the field constant (**final**) and the compiler verifies values are assigned by each constructor, and cannot be changed after.

Different types of methods

Class methods can be:

- Constructors: initialize object fields
- Mutator & Accessor methods:
  - Mutator/Setter: control and verify changes in a field. Often have the form `setX()`
  - Accessor/Getter: return the value of a private field (usually accompanies a setter method)
- Business methods:
  - …: perform other needed calculations and actions

Setters/Getters

```
class Point{
  private double x;
  private double y;

  public Point (double x, double y){
    this.x = x;
    this.y = y;
  }
  public double getX (){
    return x;
  }
  public void setX (double x){
    this.x = x;
  }
  public double getY (){
    return y;
  }
  public void setY (double y){
    this.y = y;
  }
}
```

Methods that allow access to (read and modify) fields …

```
Point p = new Point (1.0,1.0);
System.out.println("%d %d\n",
          p.getX(), p.getY() );
```


Setters/Getters

```
class Point{
  private double rho;
  private double theta;

  public Point (double x, double y){
    init(x,y);
  }
  private void init (double x, double y){
    rho = hypot(x,y);
    theta = atan2(x,y);
  }
  public double getX (){
    return rho*cos(theta);
  }
  public void setX (double x){
    init(x,getY());
  }
  public double getY (){
    return rho*sin(theta);
  }
  public void setY (double y){
    init(getX(),y);
  }
```

Methods that allow access to (read and modify) fields …

And change the implementation, without changing the method call

```
Point p = new Point (1.0,1.0);
System.out.println("%d %d\n",
          p.getX(), p.getY() );
```

Don’t overdo it

Don’t put setters/getters for everything

- A getter means more code to maintain, so add them only when needed to see a field
- A setter is only needed if we are sure we need to change the field of an object
- Limit the number of objects with mutators, i.e. objects that can change (mutable vs immutable objects)

Exercise 2

www.lri.fr/~anab/teaching/remedialJava/ex-session2.pdf

Reminder:

- Visibility keywords private < protected < public
- Members of a class (fields, methods) should be private unless necessary
- Setter/getter methods can help access private fields if needed

- secours
- fkns922j

Notes on style

Some notes on style

- Use good/expressive names for variables

```
String a1;
int a2;
double b;          // BAD!!

String firstName; // GOOD String lastName;
// GOOD
int temperature;     // GOOD
```

Some notes on style

- Use good/expressive names for variables
  - Differentiate field variable names from other variables

```
public class TimeOfDay{
    private int _hour;
    private int _minute;

    public TimeOfDay(int hour, int minute){
        _hour = hour;
        _minute = minute;
    }
}
```


Some notes on style

- Use good/expressive names for variables
  - Differentiate field variable names from other variables
  - Java naming conventions:
    - Variables: Nouns, lowercase first letter, capitals separating words
      e.g., `x, shape, highScore, fileName`
    - Methods: Verbs, lowercase first letter
      e.g., `getSize(), draw(), drawWithColor()`
    - Classes: Nouns, uppercase first letter
      e.g., `Shape, WebPage, EmailAddress`

Some notes on style

- Use good/expressive names for variables
  - Differentiate field variable names from other variables
- Use indentation
- Use whitespaces in complex expressions

Some notes on style

- Use good/expressive names for variables
  - Differentiate field variable names from other variables
- Use indentation
- Use whitespaces in complex expressions
- Do not duplicate tests

Some notes on style

- Do not duplicate tests

```
if (basePay < 9.0) {
    ...
} else if (hours > 60) {
    ...
} else if (basePay >= 9.0 && hours <= 60) {
    ...
}
```


Some notes on style

- Do not duplicate tests

```
if (basePay < 9.0) {
    ...
} else if (hours > 60) {
    ...
} else {
    ...
}
```

Exercise 3 - Homework

www.lri.fr/~anab/teaching/remedialJava/ex-session2.pdf

Reminder:

- Consider encapsulation: private fields (and methods unless needed), use of getter/setter methods
- Remember naming conventions/styles (differentiate field `_variables`, name variables as `lowerCaseNouns`, and methods as lowerCaseVerbs())
- The `static` keyword creates fields/methods that are common for all objects of a class

A quick OOP summary

OOP – Object (1)

1. Everything is an object.

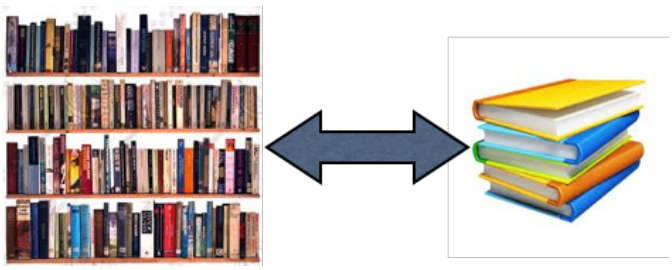
- An object is an improved variable: it stores data, we can query it, request that it acts upon itself.
- We can take any concept of a problem we want to solve (a book, a dog, an administrative service,…) and represent it as an object in the program




OOP – Object (2)

2. A program is a collection of objects that tell each other what to do, by passing messages.

- To make a request to an object, we pass it a message. More specifically, a request can be seen as a call to a **public** method/function that belongs to the target object.



OOP – Object (3)

3. Each object has its own dedicated memory space, made up of other objects (attributes)

- A new object is made up by other existing objects (or collections of them). This way, the complexity of a program is hidden in the simplicity of the role of each object.

OOP – Object (4)

4. Each object is of a specific type.

- In OOP, each object is an instance of a class (can be considered as a type, or a pattern). The most important characteristic of a class is:
  - What messages can we send to it?

OOP – Object (5)

5. All objects of the same type (class) can receive the same messages.

- An object of the type “circle” is also un object of the type “geometric form”, so a circle can accept messages destined to geometric forms.