Introduction to OO Programming

- With the help of the JAVA LANGUAGE & Object Oriented DESIGN

Step 3

Inheritance
Polymorphism

COURSE OVERVIEW

- Inheritance

- Polymorphism

- Abstract classes

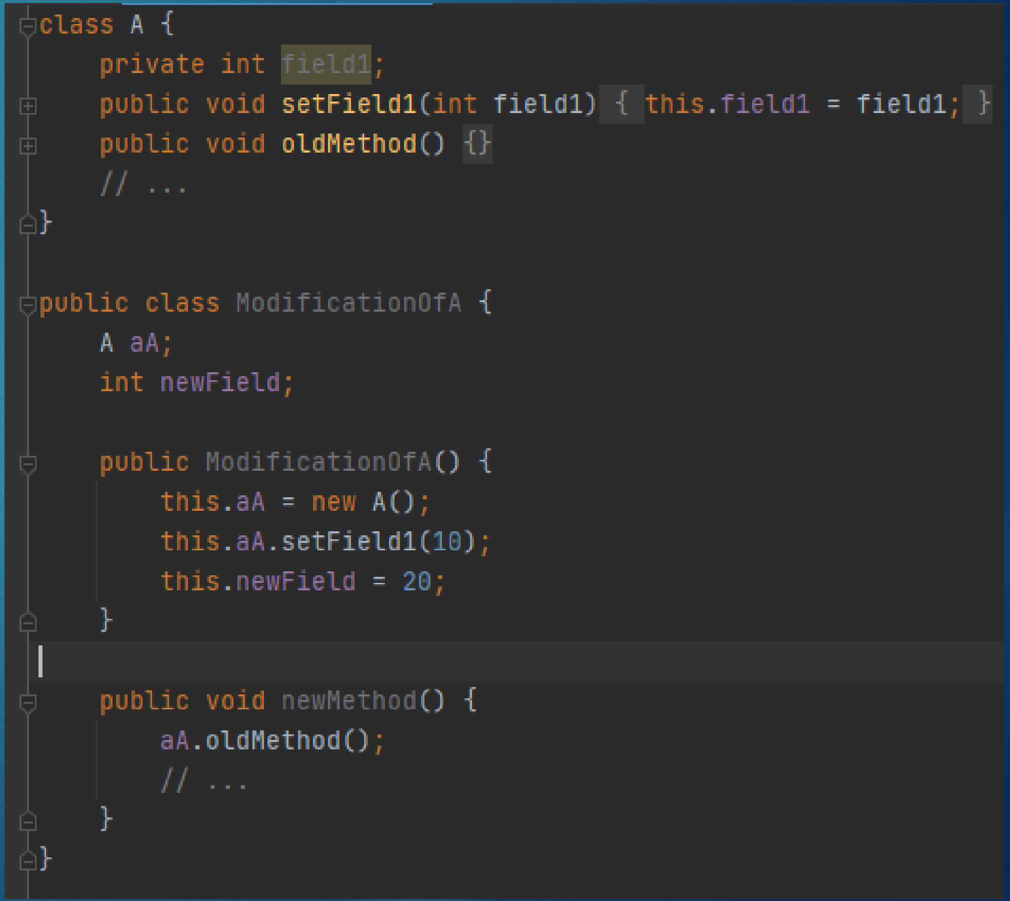
1 – Inheritance

Reuse?

For a new application, we (sometimes) do not want to reinvent the wheel. Sometimes we want to add functionalities, want to modify a process…

Solution :

1. Copy/Paste then modify the used class (bad idea)
2. Add a relationship in a bigger class – reuse-by-composition (not so bad, but still in some cases…)



```java
class A {
    private int field1;
    public void setField1(int field1) { this.field1 = field1; }
    public void oldMethod() {}
    // ...
}

public class ModificationOfA {
    A aA;
    int newField;

    public ModificationOfA() {
        this.aA = new A();
        this.aA.setField1(10);
        this.newField = 20;
    }

    public void newMethod() {
        aA.oldMethod();
        // ...
    }
}
```

Inheritance

THE "IS-A Relationship"

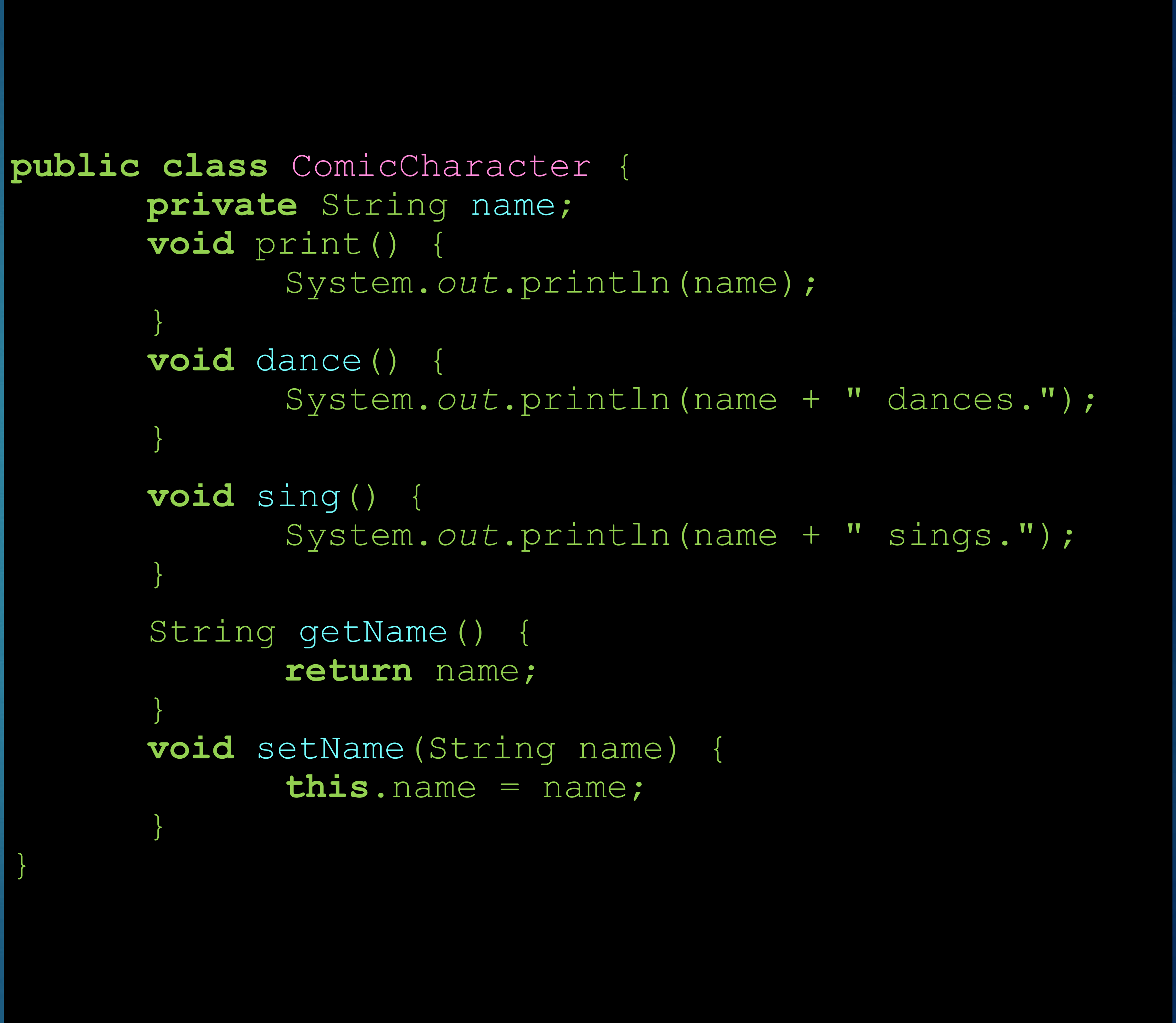
- An object of a subclass is also an object of the superclass (“*is-a*" relationship)
  - Example: superclass Vehicle, subclass Truck: “A truck is a vehicle…”
- An object of the subclass can be used anywhere where an object of the superclass can be used (=substitution); inaccessibility of attributes

- In addition to the characteristics of the superclass, the declaration of the subclass adds new characteristics (fields, methods)
- A subclass is always a specialization of the superclass
- Layers of abstraction ⇔ hierarchy of classes

Reuse

A special keyword "**extends**", a Java class can be extended :

- Subclass extends SuperClass {
  … // some extra features
  }



```
public class ComicCharacter {
      private String name;
      void print() {
            System.out.println(name);
      }
      void dance() {
            System.out.println(name + " dances.");
      }

      void sing() {
            System.out.println(name + " sings.");
      }

      String getName() {
            return name;
      }
      void setName(String name) {
            this.name = name;
      }
}
```

First inheritance…

they are back…

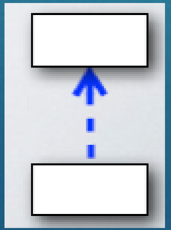
```
public class SuperHero extends ComicCharacter {

        // inherits characteristics from ComicCharacter ('print', 'dance',
'sing'),
        // adds fighting functionality:
        protected String superPower;

        void fight() {
                System.out.println (getName()+" fights.");
        }
        String getSuperPower() {
                return superPower;
        }
        void setSuperPower(String superPower) {
                this.superPower = superPower;
        }

}
public class FlyingHero extends SuperHero {
        // ComicCharacter behavior extended with the flying functionality
        void fly() {
                System.out.println(getName()+" flies.");
        }
```

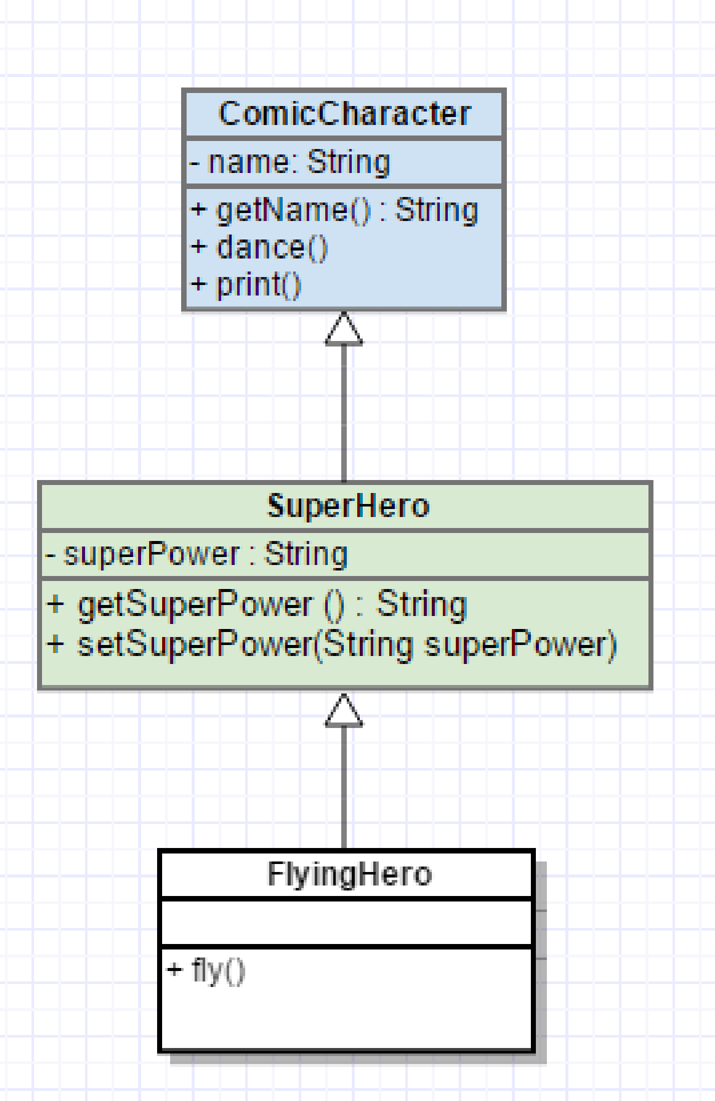
1 – Inheritance

Inheritance's worth it !

- A class which "extends" another, inherits some properties from its ancestors.
  - **Inheritance is transitive**
  - We also say that the subclass "***specializes***" its super class
  - It can **directly access** to some fields and methods from its ancestors (at least the ones defined as accessible i.e not private)
  - This is a **dependency** between classes !

To check





Visibility (access power – or not)

Direct Use

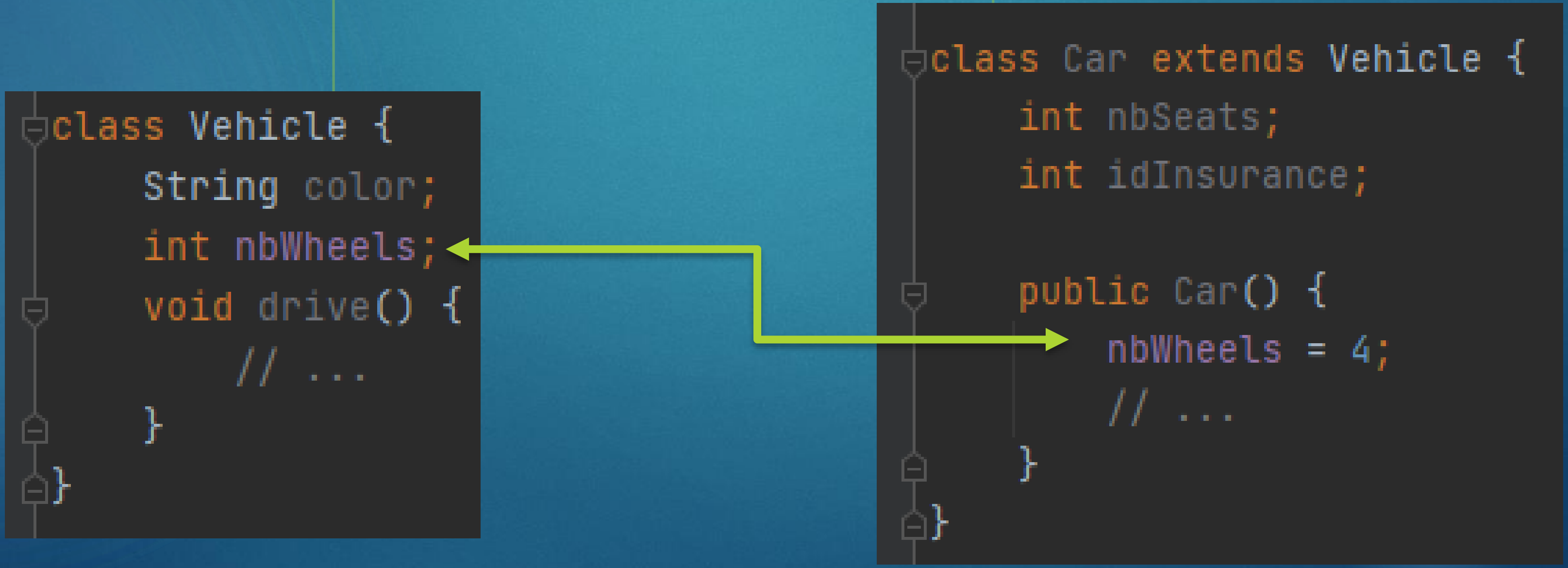
All **public** fields may be used by subclasses

No access

It can't use **private** members (must use accessors)

Just you (and others)

A class may give access only to its subclasses with the **protected** keyword. But it gives access to all classes of the same *package* as well, so **you may want to avoid it** (next course)



```
class Vehicle {
    String color;
    int nbWheels;
    void drive() {
        // ...
    }
}
```

```
class Car extends Vehicle {
    int nbSeats;
    int idInsurance;

    public Car() {
        nbWheels = 4;
        // ...
    }
}
```

Possible or not? (in JAVA)

1 – Inheritance

- When writing a subclass : NEVER EVER EVER declare the same fields.
- An object from a subclass can be **substituted** wherever its ancestors may be expected (nice huh?)
- A subclass can (re)declare the same method name as was declared in some of its ancestors, usually to add some specific processing. This is called "**overriding**"
- Moreover, a subclass can reuse an ancestor's method inside an overridding method (very nice….)
- A parent class may prevent subclasses to override one of its methods with the **`final`** keyword
- Static members (fields and methods) are inherited

First inheritance…

they are back…

```
public class SuperVillain extends SuperHero {

      // inherits characteristics from ComicCharacter
      // adds the functionality to 'fight'
      protected String superPower;

      void dance() {
           System.out.println("Sorry, but villain doesn't dance
!!!");
      }
}
```

Inheritance and constructors

1 – Inheritance

- Remember: JAVA defines a default constructor (without params) **if and only if** you don't define one in your class…

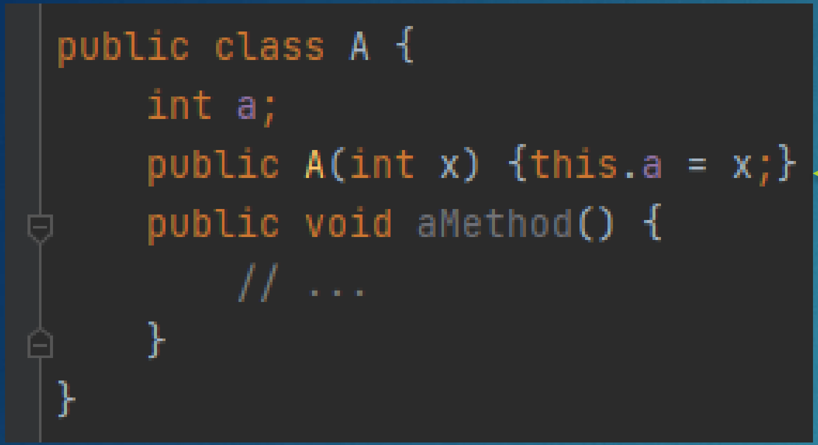
- In a constructor, you can call one of the parent’s constructor with **`super`**`(arg1, arg2,…)`

- This "*super*" statement (in JAVA) **must be the first** in your overridden constructor

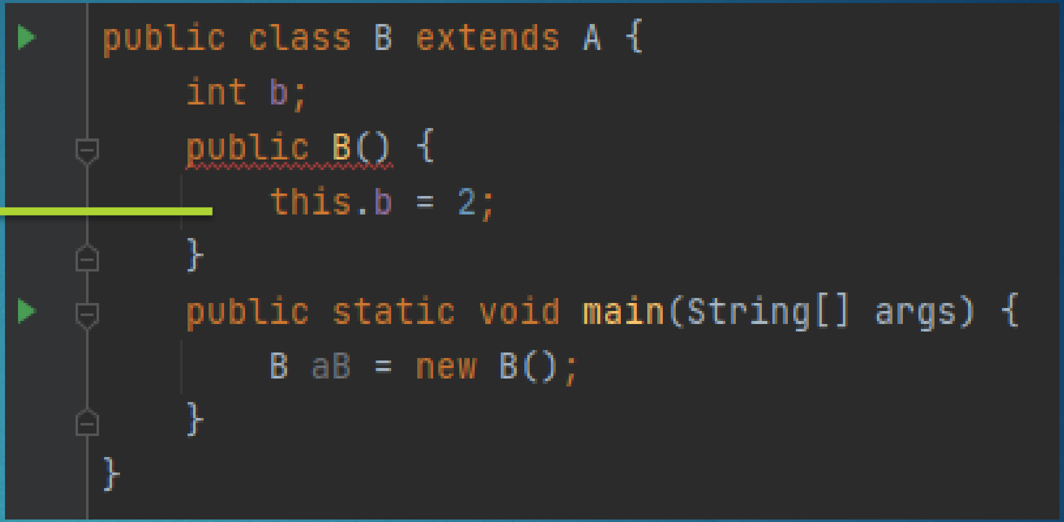
- JAVA always implicitly calls the default constructor of the superclass, unless you do call a specific constructor of the superclass

- The "***super***" keyword can be used inside any overriding method assuming one ancestor class declares it as "public" or "protected"

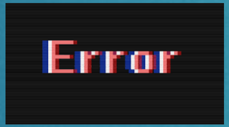
Inheritance and constructors



```
public class A {
    int a;
    public A(int x) {this.a = x;}
    public void aMethod() {
        // ...
    }
}
```

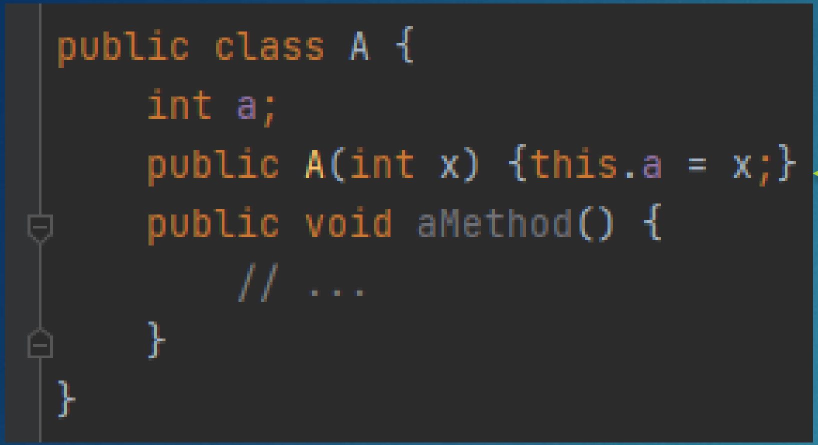


```
public class B extends A {
    int b;
    public B() {
        this.b = 2;
    }
    public static void main(String[] args) {
        B aB = new B();
    }
}
```

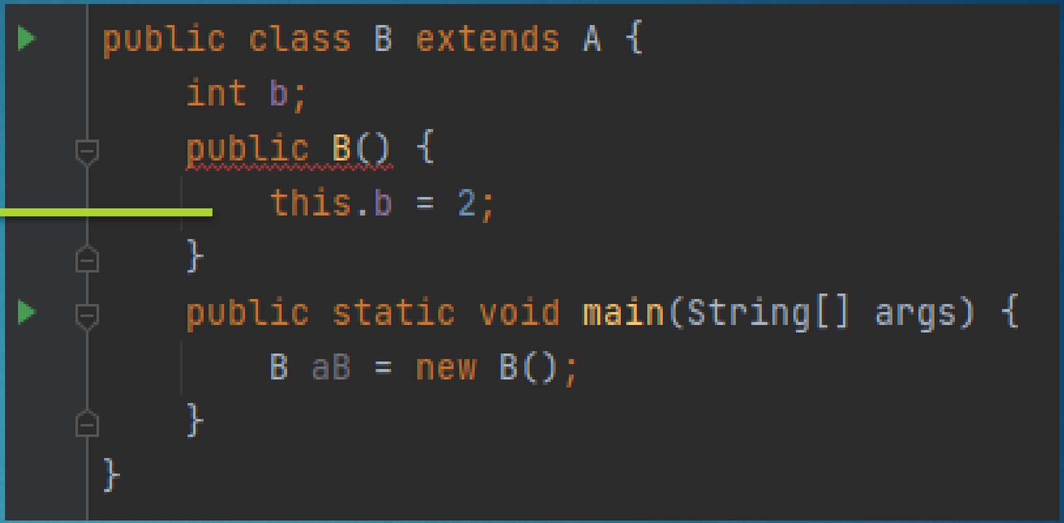


1 – Inheritance

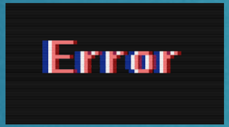
Inheritance and constructors



```
public class A {
    int a;
    public A(int x) {this.a = x;}
    public void aMethod() {
        // ...
    }
}
```



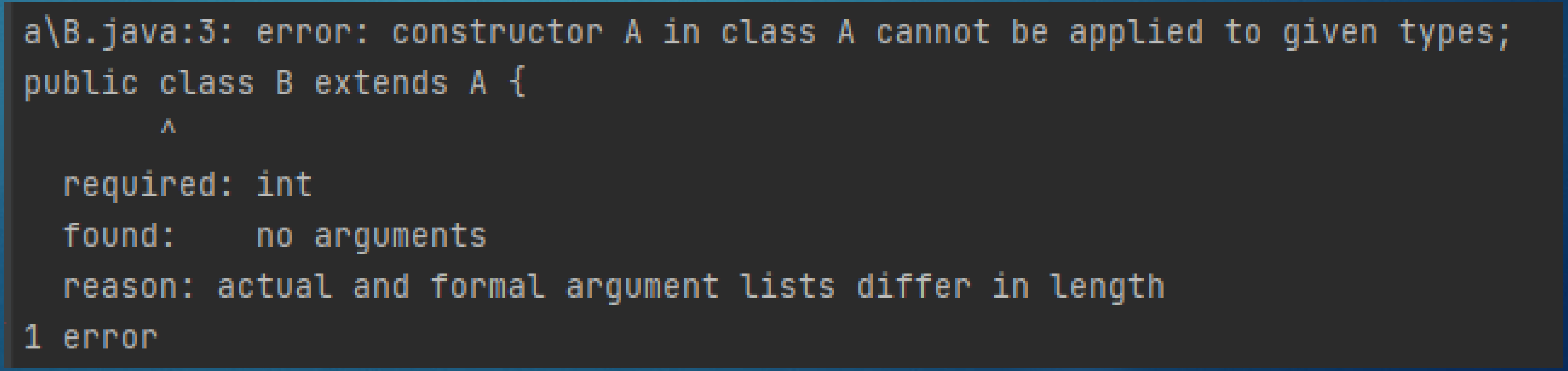
```
public class B extends A {
    int b;
    public B() {
        this.b = 2;
    }
    public static void main(String[] args) {
        B aB = new B();
    }
}
```



Error

In B's constructor, there's an implicit call to constructor A(), which does not exist! (only one with an int param) !

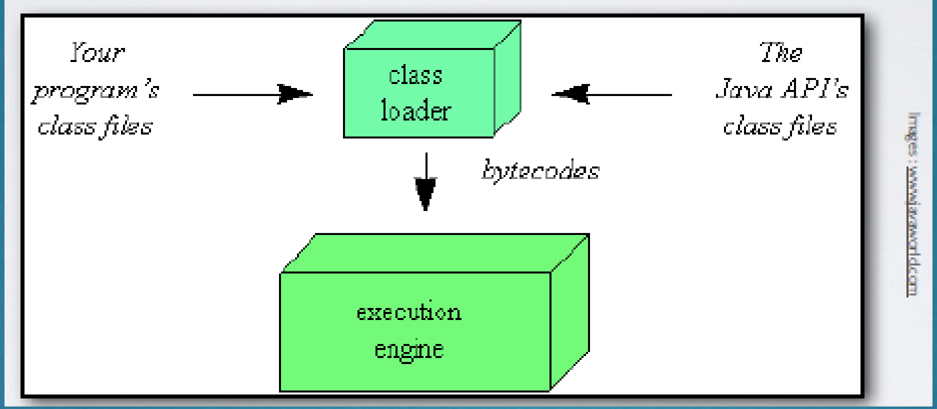
Remember this message, you'll see it !



```
a\B.java:3: error: constructor A in class A cannot be applied to given types;
public class B extends A {
       ^
  required: int
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

About memory

- At runtime, the JRE class loader piles classes as and when declared in your executed code



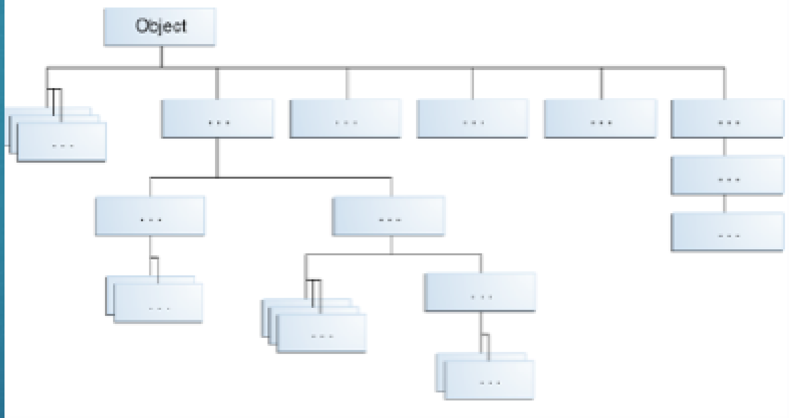
Images : www.javaworld.com

- Constructors are called when instances are piled up in memory, sometimes in cascade.

1 – Inheritance

NOTES

- Every class inherits from the "mother of all" class **`Object`** (no need to specify the `extends`). All classes therefore gain access to the famous **`toString()`** method
- Inheritance creates a hierarchy in your class model



- Fortunately, **JAVA doesn't allow multiple inheritance** (Python and C++ do, but if ever, use it carefully)

2 – Polymorphism

BE TAKEN FOR ANOTHER

> “In biology, polymorphism means that a species can take multiple forms”

DEPENDING ON THE MOMENT, YOU CAN TURN INTO A DRIVER, A WORKER, A CLIENT, …, OR JUST A NORMAL HUMAN BEING

In OOP, a same functionality can ben realized (implemented) in different places, in many different ways depending on the precise object that does it

“Now Speak!”
“Quack”
“Woof”
“Meow”

POLYMOPHIC INHERITANCE

you see me… that's not (exactly me)

- *Sending a message to an object, doesn't mean that you really know the real type of the object, just that it references a method to realize a process according to the message*
- An example on http://codeboard.io/projects/14556 ((re)create an account firstnameLastname with your email)

OVERLOADED?

- *A subclass can rewrite (completely or not) some methods issued from its parent classes:*
  this is "overriding"
- *Main interest is to precise a service according to some constraints the subclass may have, or needing more parameters to be executed (well)*

Example: access to parent's overriden method

```
public class Vehicle {
        //... blabla
        protected int position;
        protected int speed;
        public void roll() throws Exception {
                position += speed;
        }
        /// etc....
}

public class Plane extends Vehicle {
        /// ...
        int altitude;
        public void roll() throws Exception {
                if (altitude > 0) {
                        throw new Exception("Plane can't roll while flying !!!");
                } else {
                        super.roll();
                }
        }
}
```

OVERLOAD

2 – Polymorphism

- Overloading may also consist to give access to same method but with different parameters
- Subclasses may use both overriding / overloading

```
public class Printer {
        //... blabla
        public void print(int i)  { … }
        public void print(double d)  { … }
        public void print(String mess)  { … }
        // etc....
        // you can change return type as well, but not a good practice !!!

        public String print(String message, String last)  { … }
}
```

NOTES

- JAVA verifies objects type during compilation, and must determine at this moment (so before execution) if a message can be actually sent or not to an object

- Anytime, it's possible to check how JAVA considers an object (who are you ?):
  - "`a instanceof B`" returns true if and only if a is from class B, (or one from one of its subclasses)
  - "`o.getClass()`" returns o's class (parameterized class, wait for next courses)

EXERCICE

- Suppose we have the following classes:

```
public class A {
            private int i;
            public A(int x) { i = x; }
            public String whoAreYou (){return "I’m an A";}
            public String toString() { return "i = " + i;}
            public String introduceYourSelf(){ return whoAreYou() + toString(); }
}// End class A
public class B extends A {
            private int j;
            public B(int x, int y) {
                        super(x);
                        j = y;
            }
            public String whoAreYou (){return "I’m a B";}
            public String toString(){return super.toString() + "\n j = " + j;}
} // End class B
public class C extends B {
            // No additional fields
            public C(int x, int y) { super(x,y);}
            public String whoAreYou (){return "I’m a C";}
} // End class C
```

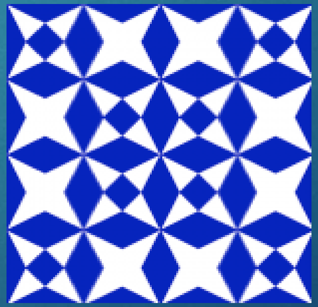
Describe what happens during the execution of: C obj = new C(5,10);

What about? `System.out.println(obj.introduceYourSelf());`

What if variable obj is of type A? (A obj = new C(5,10);)

3 – Abstract classes

WON'T DO ALL THE JOB !



SHARING CODE PARTS

SOMETIMES NEEDED

- Many classes A,B,C… may share processes (methods and code), but none of them is really the parent of one another

- So why not create a base class "P" for all of them, and place the common code in it?! (Developers are – very – lazy!)

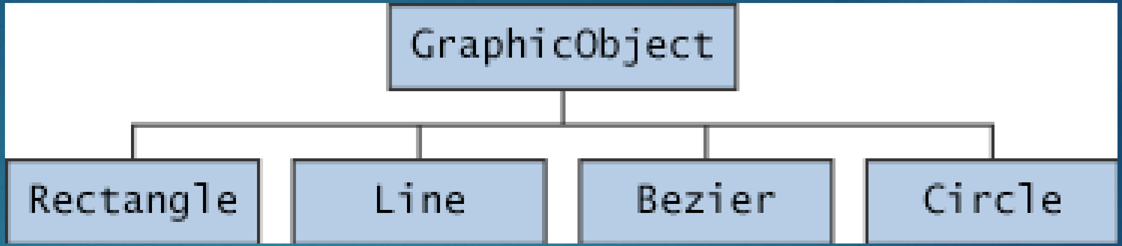
BUT !!! NONSENSE !

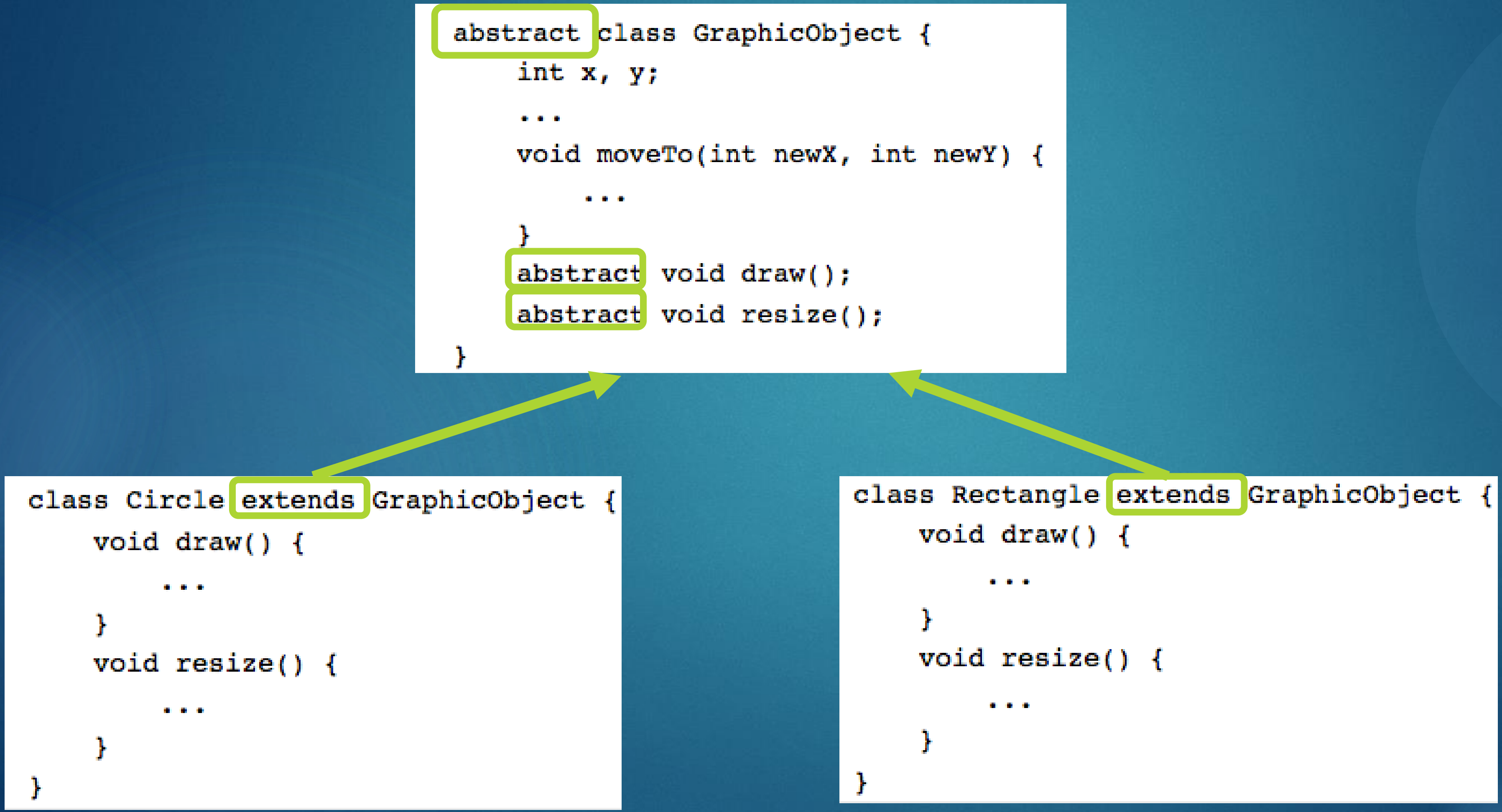
- P may lack some code to run as expected?! So creating a P instance is nonsense !!!

- No problemo ! Declare P as "`abstract`", so nobody will be able to create a (real) instance of P

- Furthermore ! We can define in P some "`abstract`" methods to force subclasses to define them, or to be abstract as well…

Example





```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

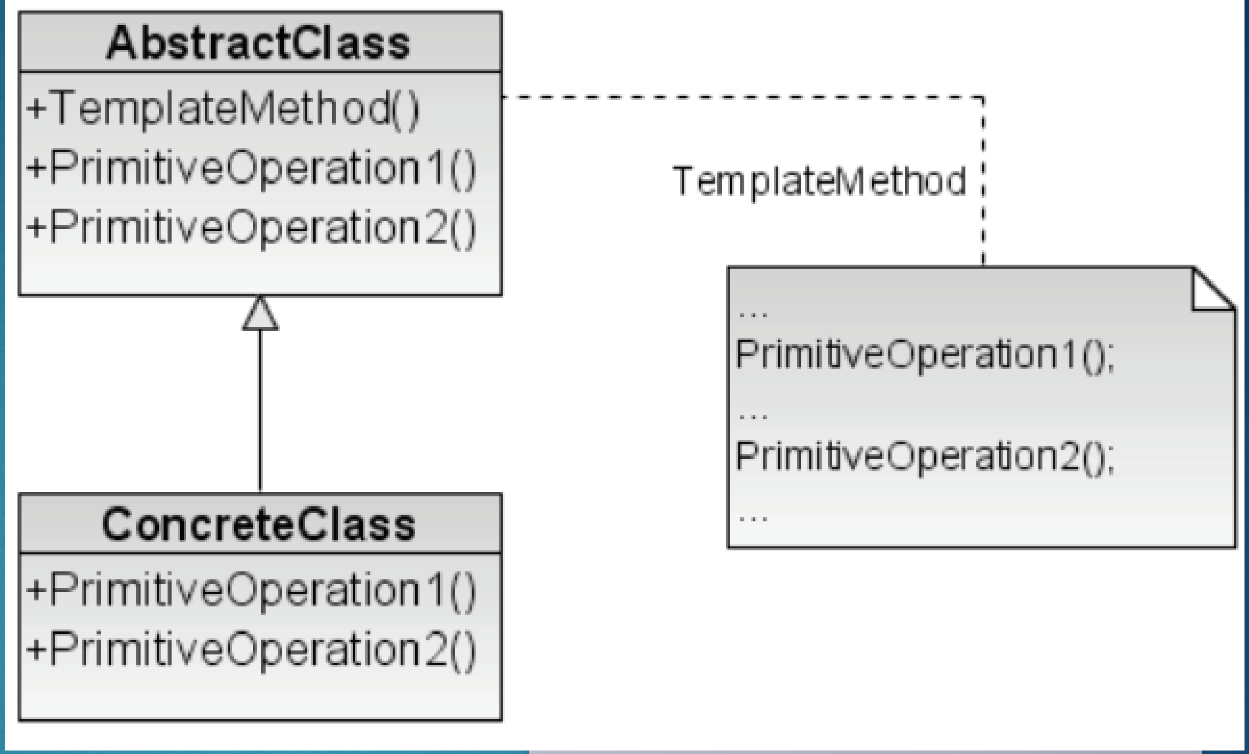
```
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

3 – Abstract classes

DESIGN PATTERNS

- There are not so many different ways to design

EXAMPLE



AbstractClass

+TemplateMethod()
+PrimitiveOperation1()
+PrimitiveOperation2()

TemplateMethod

...
PrimitiveOperation1();
...
PrimitiveOperation2();
...

ConcreteClass

+PrimitiveOperation1()
+PrimitiveOperation2()

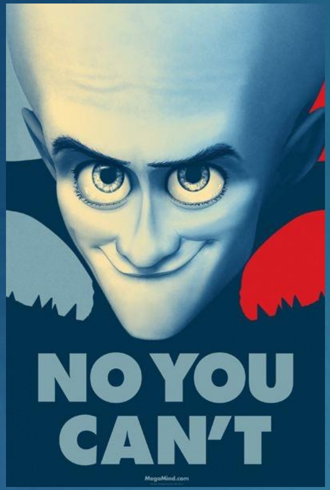
Last example uses a design pattern "Factory", or "template method".

- Parent class defines a pattern for an algorithm (`moveTo()` method) using abstract methods (for now)
- Parent class orders its future subclasses to define the `'draw()'` method needed by moveTo… according to their specificity

https://en.wikipedia.org/wiki/Software_design_pattern#Creational_patterns

ABSTRACT RULES

3 – Abstract classes



NO STATIC

You can't for methods, I said no! you can't! Why???

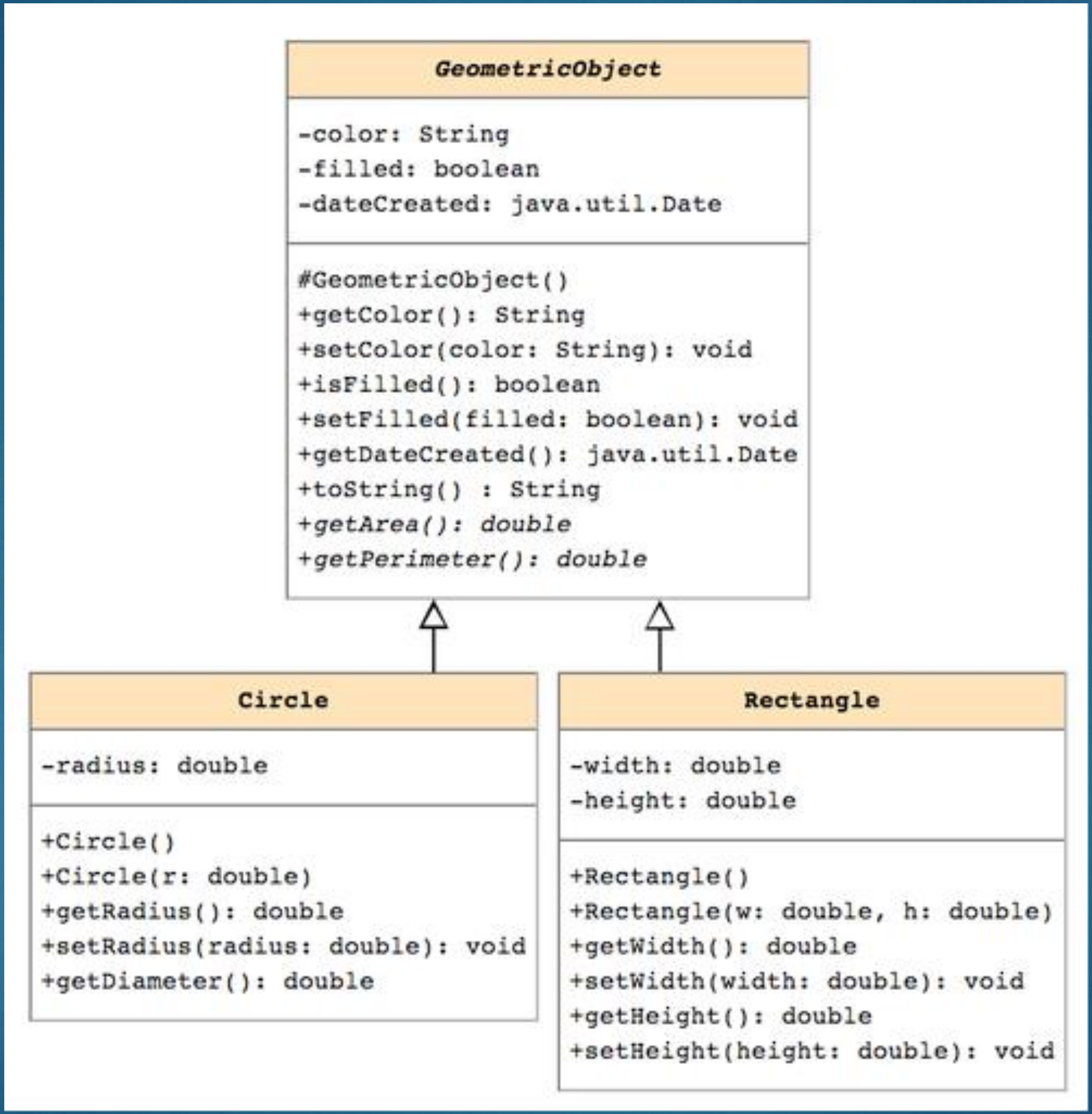
Fields

Abstract class can define fields, even static!

UML

In UML, simply write class / methods in ***italic (not in green 🖳)***

3 – Abstract classes



GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

#GeometricObject()
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString() : String
+getArea(): double
+getPerimeter(): double

Circle

-radius: double

+Circle()
+Circle(r: double)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(w: double, h: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

ANOTHER EXERCICE

- Consider the evaluation of arithmetic expressions with the four operators + - * /
- An expression is defined recursively as follows:
  - Either it’s a constant (1.5 for ex.)
  - Or it’s a “complex" expression with the following form: a op b
  - where a and b are expressions and op is one of the 4 operators
- **Write the Java classes (and UML class diagram) which enable to build and evaluate expressions so that we can write (in a main method for ex.):**

```
Constante a = new Constante (5);
Constante b = new Constante (2);
Constante c = new Constante(3);
ExpressionComplexe e1 = new ExpressionComplexe (a, '+', b);
ExpressionComplexe e2 = new ExpressionComplexe (e1, '*', c);
ExpressionComplexe e3 = new ExpressionComplexe(new Constante(4), '*', e2);
System.out.println(a.eval());  // 5.0
System.out.println(e1.eval()); // 7.0
System.out.println(e2.eval()); // 21.0
System.out.println(e3.eval()); // 84.0
```

Now, it's your turn…
Lab Session 3 (Pokemon classes)

- Pokemons are friendly animals who are passionate about OOP. There are four main categories of pokemon:
  - Sports Pokemons
  - Stay-at-home Pokemons
  - Sea Pokemons
  - Cruising Pokemons

This is the end…for now

Now, it’s your turn…
Lab Session 3 (Pokemon classes)

- Pokemons are friendly animals who are passionate about OOP. There are four main categories of pokemon:
  - Sports Pokemons: characterized by a name, a weight(in kg), a number of legs, a size (in meters) and a heart rate measured in number of beats per minute. These pokemons move on the earth at a certain speed that can be calculated as follows: speed = number of legs * size * 3
  - Stay-at-home Pokemons: characterized by a name, a weight (in kg), a number of legs, a size (in meters) and the number of hours per day during which they watch TV. These pokemons also move on the earth at a certain speed that can be calculated as: speed = number of legs * size * 3

This is the en

Now, it’s your turn…
Lab Session 3 (Pokemon classes)

- Pokemons are friendly animals who are passionate about OOP. There are four main categories of pokemon:
  - Sea pokemons: characterized by a name, a weight (in kg) and a number of fins. These pokemons only move in the sea at a speed that can be calculated as follows:
    speed = weight / 25 * number of fins
  - Cruising pokemons: characterized by a name, a weight (in kg) and a number of fins. These pokemons only move in the sea at a speed that can be calculated as:
    speed = (weight / 25 * number of fins) / 2

This is the e

Now, it’s your turn…
Lab Session 3 (Pokemon classes)

- For each of these four categories of pokemon, we want to have a method toString () which returns (in a string) the characteristics of the pokemon
- For example, the toString () method invoked on a sports pokemon would return: "I am the pokemon Pikachu, my weight is 18 kg, my speed is 5.1 km / h, I have 2 legs, my size is 0.85m, my heart rate is 120 beats per minute “
- When invoked on a stay-at-home pokemon it could return: "I am the pokemon Salameche, my weight is 12 kg, my speed is 3.9 km / h, I have 2 legs, my size is 0.65m, I watch TV 8 hours a day “
- On a sea pokemon: "I am the Rondoudou pokemon, my weight is 45 kg, my speed is 3.6 km / h, I have2 fins “
- On a cruising pokemon: "I am the Bulbizarre pokemon, my weight is 15 kg, my speed is 0.9 km / h, I have 3fins "

This is the end…for now

Now, it’s your turn…
Lab Session 3 (Pokemon classes)

- Define a UML class diagram then write the Java code corresponding to the classes described above
- Write a class allowing the manipulation of a collection of pokémons (stored in an array)
- Add to this class methods allowing to empty the collection, add a pokemon, calculate the average speed of all pokemons and the average speed of sports pokemons
- Also implement the method toString()
- Write a test class for the previous classes

This is the