R2.01 : Object-oriented development (OOD)

Coordinator : Isabelle Blasquez

My name: Cristina Onete

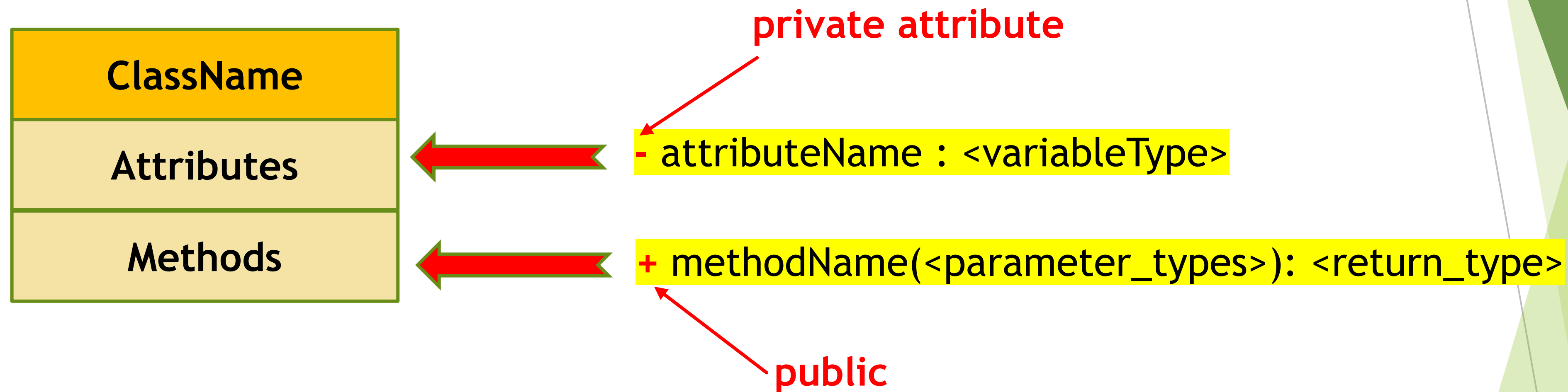
cristina.onete@gmail.com

Slides : https://www.onete.net/teaching.html

Brief reminder:
class diagrams

A means of summarizing a class

- A class diagrams indicates the class attributes and method signatures



- Why would we want to represent classes in this fashion?

To get some perspective on the code

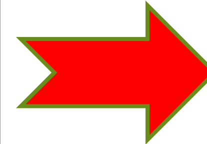
To precisely specify what the code does

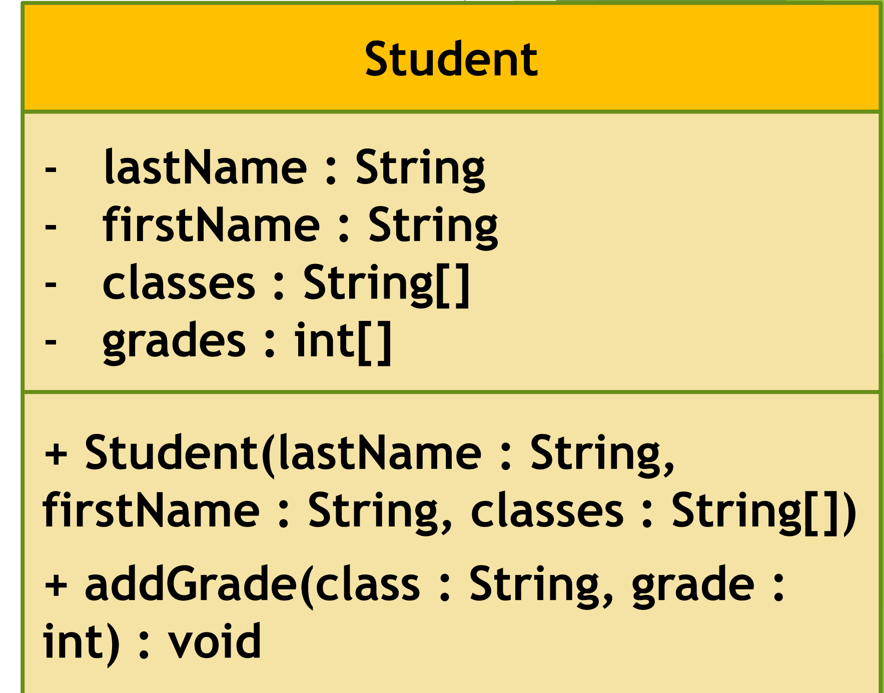
To better design one's code

To understand how to properly test the code

Example

```
public class Student{
    private String lastName;
    private String firstName;
    private String[] classes;  // student's classes
    private int[] grades; // student's grades

    public Student(String lastName, String
firstName, String[] classes){
        this.lastName = lastName;
        this.firstName = firstName;
        this.classes = classes;
        grades = new int[classes.length];
    }

    public void addGrade(String class, int grade){
        // adds new grade for target class
    }
}
```



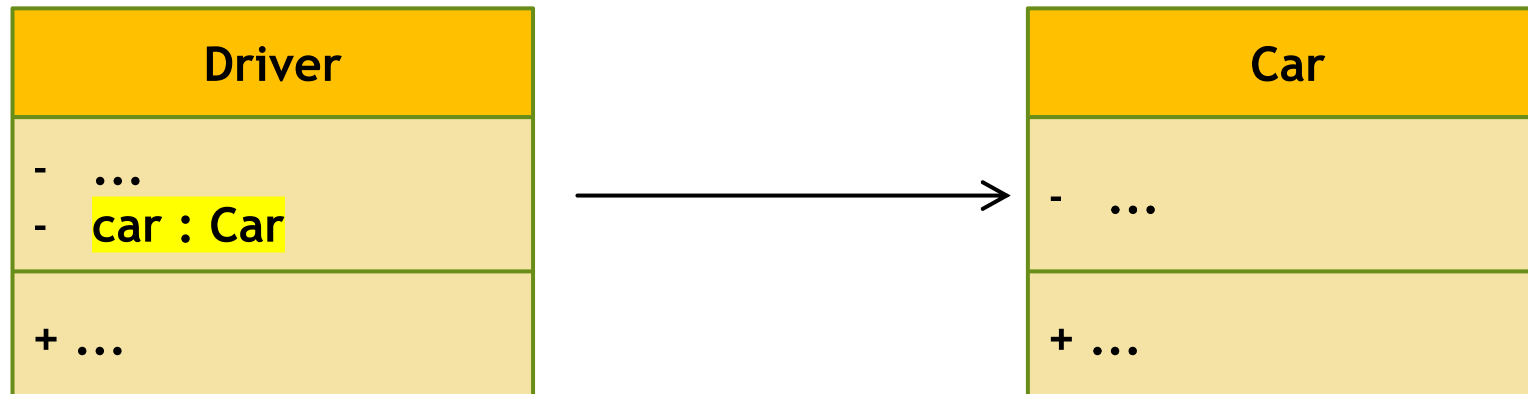


Class association: "has a"

- Example: a driver has (owns) a car

```
public class Driver{
    ...
    // other attributes: name,
licenseType, license...
    Car car;

    ...
    // constructors, other methods...
}
```

```
public class Car{
    ...
    // attributes : make, model,
licensePlateNumber

    ...
    // constructors, other methods
}
```

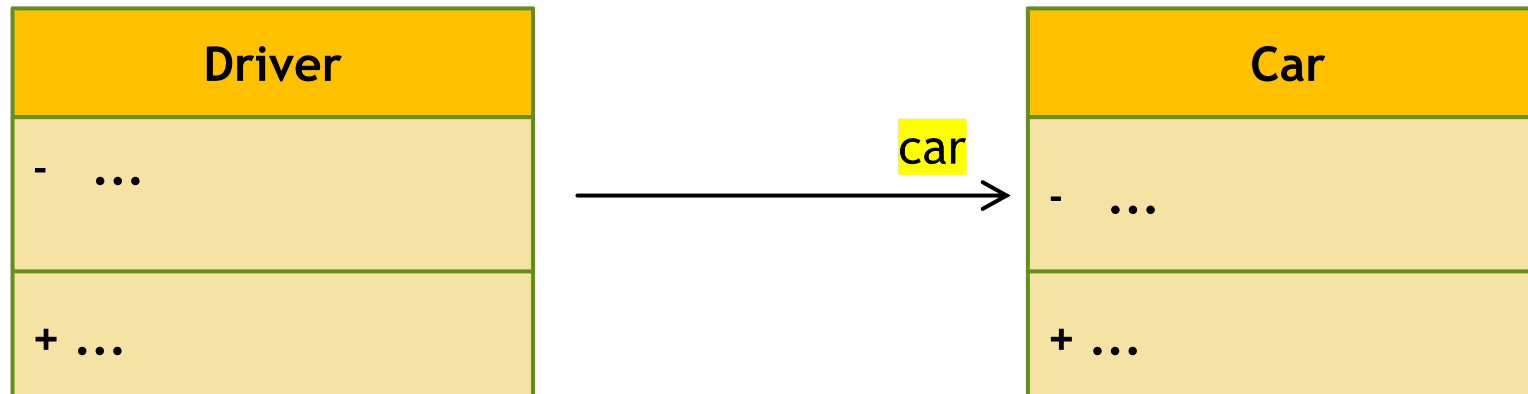


Class association: "has a"

- Example: a driver has (owns) a car

```
public class Driver{
    ...
    // other attributes: name,
licenseType, license...
    Car car;

    ...
    // constructors, other methods...
}
```

```
public class Car{
    ...
    // attributes : make, model,
licensePlateNumber

    ...
    // constructors, other methods
}
```



Two-way (mutual) association

- A driver has a car, and a car has a driver

```
public class Driver{
    ...
    // other attributes: name,
licenseType, license...
    Car car;

    ...
    // constructors, other methods...
}
```

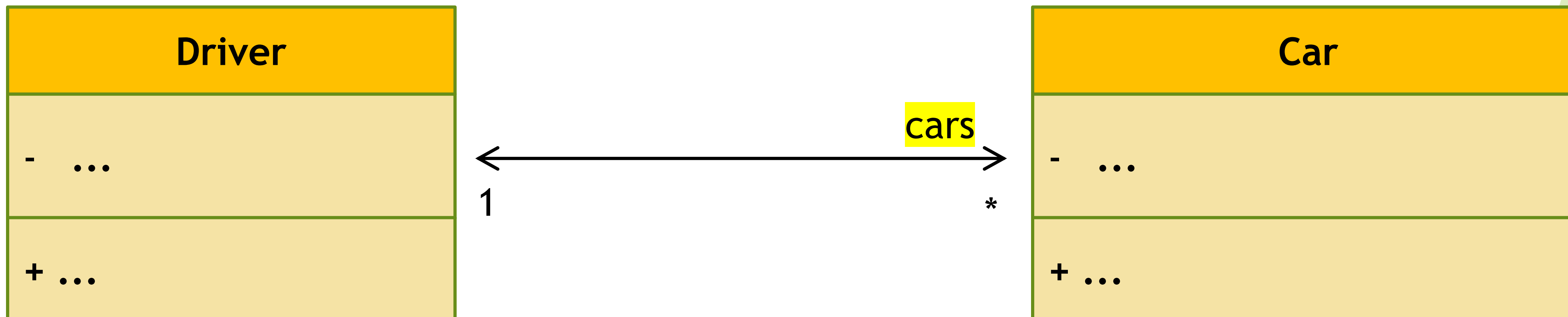
```
public class Car{
    ...
    // attributes : make, model,
licensePlateNumber
    Driver owner;

    ...
    // constructors, other methods
}
```

Multiplicity in association

- How many objects of TypeA does an object of TypeB have?

```
public class Driver{
    ...
    // other attributes: name,
licenseType, license...
    Car[] cars;

    ...
    // constructors, other methods...
}
```

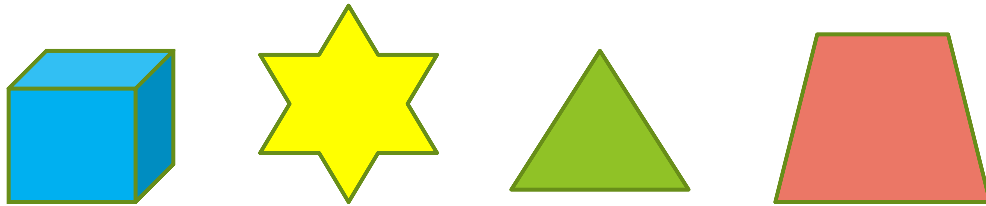
```
public class Car{
    ...
    // attributes : make, model,
licensePlateNumber

    ...
    // constructors, other methods
}
```



Inheritance

Objects with similar characteristics

- Take the example of these blocks:
  - They can be inserted within the larger cube, moved, etc.
  - Let's consider the following shapes

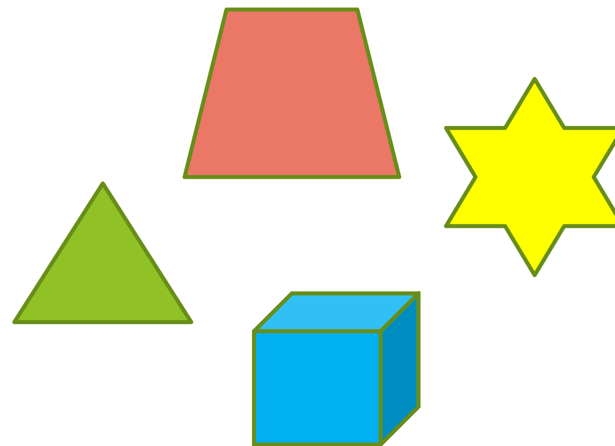


  - One can build by turning those shapes around
  - Say some shapes also come with bells and make special sounds

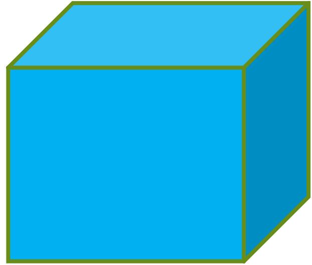
= CLING CLING

Objects with similar characteristics

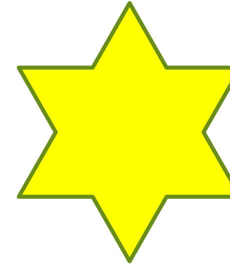
- Similar characteristics
  - They are all types of blocks
  - They come in shapes
  - They can be turned around
- Different characteristics
  - The shapes differ
  - Some blocks make sounds
- Code refactoring: bunch similar characteristics together



Without inheritance



```
Class Cube{
    ...
    public void turnAround (int x){
    // turns cube around by x degrees
    ...
    }
}
```



```
Class Star{
    ...
    public void turnAround (int x){
    // turns cube around by x degrees
    }
    public void makeSound(){
    // makes sound CLING CLING
    }
}
```

Duplicating code

- ... is a really bad idea
- For instance: the method `turnAround(int x)` in classes Cube and Star
- Why a bad idea ?

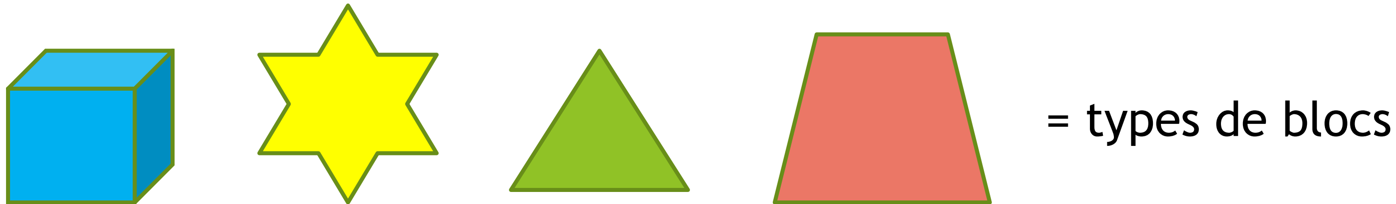
New code **might be incompatible** with old code (think attributes, variables...)

Multiple instances of a copied bug must be **debugged and resolved** separately

Useless: An instance of the copied code might be solving an issue already solved in existing code

Inheritance in Java

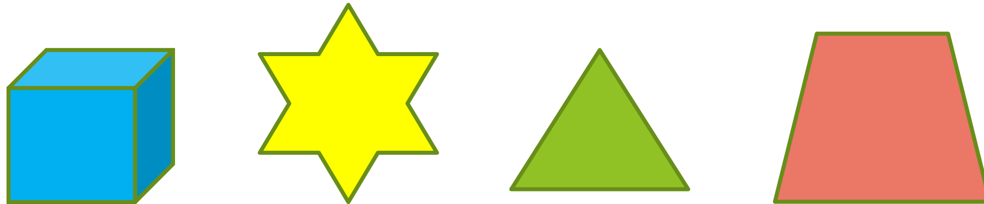
- A means of representing in code the concept "is a type of"
- Inheritance starts from a superclass
  - Which describes common characteristics:
    - Attributes: shape name, colour
    - Methods: `void turnAround (int x)`
  - Here the superclass is **Block**



- Next step: introduce subclasses inheriting from the superclass: Cube, Star, Triangle, Trapezium

Factorisation vs. conceptual differences

- Our example includes blocks with common features:



  - Attributes: each block has a shape, a colour
  - Methods: turnAround: all blocks can be turned around

Cubes, stars, etc. inherit these general block characteristics

- How about the method `void makeSound()`?

The method makeSound is specific exclusively to star blocks

Inheritance and private visibility

- In superclass **Block**:
  - Attributes: `shapeName, colour`
  - Attributes declared, but not instantiated

```
public class Block{
    private String shapeName;
    private String colour;
}
```

- **Triangle** subclass inherits from **Block**:
  - No need to mention `shapeName, colour`: they are inherited automatically
  - Attribute instantiations (in constructor)
    - `shapeName` set to "`triangle`", `colour` set to "`green`"

- Need to reference in subclass the (private) attributes of superclass

HOW ?

Private, public, protected

- Private vs. public attributes (reminder):
  - Public attributes are visible everyone in the program
  - Private attributes can only be referenced directly from within their class
  - Getters and setters are required to handle private attributes

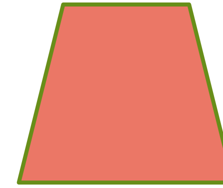
Often attributes private in superclasses (use getters/setters)
Methods are more often given protected visibility

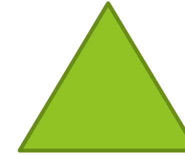
- Protected attributes (in s
  - Can be referenced directly from the class and its subclasses

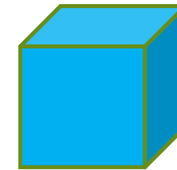
	classe	sousclasse	ailleurs
public	✓	✓	✓
private	✓	✗	✗
protected	✓	✓	✗

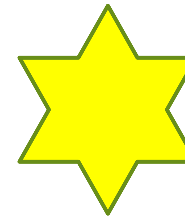
Example: superclass Block

```
public class Block{
    protected String shapeName;
    protected String colour;

    public Block(String shapeName, String colour){
        this.shapeName = shapeName;
        this.colour = colour;
    }

    public void turnAround (int x){
        // code for turning shape around
    }
}
```





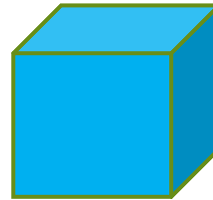




Example: subclass Cube

keyword "extends" specifies an inheritance

```
public class Cube extends Block{
    // subclasses inherit all the methods and attributes of a superclass

    // Problem #1: the constructor(s)
}
```



- A constructor of class Block bears the name of its class: Block
- A constructor of Cube must be called Cube, not Block
- Could we rely on the constructor by default (inherited from Java.lang.Object) ?
  - On the bright side: constructor already exists
  - Unfortunately: superclass Block has a constructor, which Cube inherits...
  - Thus, subclass Cube cannot use the constructor by default

Constructors in the subclass

- We use and adapt the constructor of the superclass

```java
public class Block{
    ...
    public Block(String shapeName,
String colour){
        this.shapeName = shapeName;
        this.colour = colour;
    }
}
```

```java
public class Cube extends Block{
    public Cube(){
    // call constructor in superclass
    super("cube", "blue");
    }
}
```

Keyword "super" refers to superclass

super("cube", "blue") calls constructor Block(String, String)

Every time we use the constructor of Block inside class Cube, we will use the keyword super !

Inheritance and duplication of code

```
public class Block{
    protected String shapeName;
    protected String colour;

    public Block(String shapeName,
String colour){
        this.shapeName = shapeName;
        this.colour = colour;
    }

    public void turnAround (int x){
        // code for turning around
    }
}
```

```
public class Cube extends Block{
    public Cube(){
        super("cube", "blue");
    }
}
```

Class Cube has:

- two inherited attributes: `shapeName, colour`
- a constructor, whose signature is `Cube();`
- inherited method `turnAround(int)`

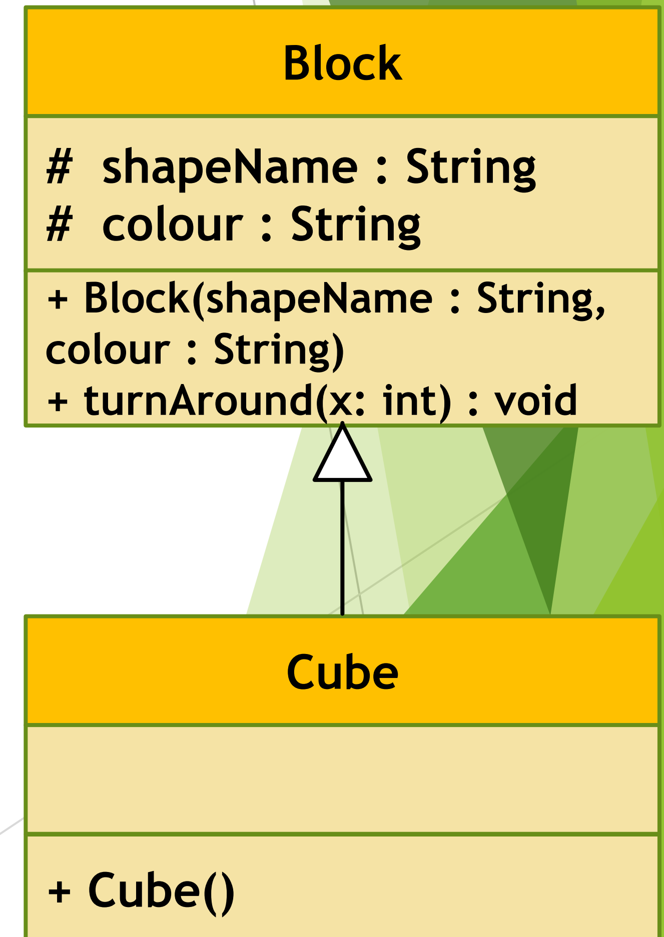
A subclass can modify a method it inherits…
But unless that is the case, the method will behave as in superclass

Inheritance in class diagrams

- Indicated by an arrow from the subclass towards the superclass
  - Inherited methods not featured in subclass, unless modified

```
public class Block{
    protected String shapeName;
    protected String colour;

    public Block(String shapeName, String colour){
        // constructor code
    }

    public void turnAround (int x){
        // code for method turnAround
    }
}
```

```
public class Cube extends Block{
    public Cube(){
        // constructor code
    }
}
```



Polymorphism

Use-case: BlockSet

- A cube is a type of block. So is a star.
- Usually, blocks come in sets
  - Depending on size, each set contains a number of blocks of each type
  - More advanced: each set has a random number of blocks of each type

- In Java block sets become a class BlockSet:
  - characterized by an attribute setSize (type char, values in 'S', 'M', or 'L')
  - an array of blocks, which can be cubes, stars, triangles, or trapeziums
    - What would be the type of this array?

Polymorphism

- *Poly* + *morphos* = many shapes
- Notion in Java that groups together objects of different types
  - cats, wolves, and people are animals
  - cubes, stars, and triangles are blocks

- Is enabled by inheritance or interfaces (next CMs)
  - Inheritance: use supertype as common type
  - A set of Block objects could contain Cube objects, Star, objects, etc.

BlockSet example

- Small block set: setSize = 'S'
  - Contains one block of each type
- Medium block set: setSize = 'M'
  - Contains 2 stars, 2 cubes, 1 triangle, and one trapezium
- Large blockset: setSize = 'L'
  - Contains 4 cubes, 3 stars, 2 triangles, 1 trapezium

Example implementation

```
public class BlockSet{
    protected char setSize; // values 'S', 'M', 'L'
    protected Block[] blocks;
    public BlockSet(char setSize){
        this.setSize = setSize;
        if (this.setSize == 'S'){
            this.blocks = {new Cube(), new Triangle(), 
new Star(), new Trapezium()};
        }
        if (this.setSize == 'M'){
            this.blocks = {new Cube(), new Cube(), new 
Triangle(), new Star(), new Star(), new Trapezium()};
        }
         if (this.setSize == 'L'){
            this.blocks = {new Cube(), new Cube(), new 
Cube(), new Cube(), new Triangle(), new Triangle(), new 
Star(), new Star(), new Star(), new Trapezium()};
        }
    }
}
```

polymorphic array

4 blocks, one of each shape

Intermezzo: enums

Java enums

- An enum is a data type in Java, specifying a set of predefined values
- For instance, Pokemon types could be stored in an enum
  - Or the setSize of BlockSet objects

```
public enum SetSize{
    S,M,L
}
```

```
public class BlockSet{
    protected SetSize setSize;
    protected Block[] blocks;

    public BlockSet(SetSize setSize){
        this.setSize = setSize;
        if (this.setSize.equals(SetSize.S)){
            this.blocks = {new Cube(), new Triangle(),
new Star(), new Trapezium()};
        }
        // ... rest of code
    }
}
```

- The alternative to using enums is checking validity of parameter

More Java enums

- Enums can be complex
- They can have attributes, contain methods, etc.

- Curious ? Have a look here :

  https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

End of intermezzo

Polymorphism and types

- Recall: objects are declared, then instantiated

```
Pokemon piplup = new Pokemon("Piplup", "WATER", 5);
```

- Polymorphic objects are also declared and instantiated
  - Crucial question: declared type and instantiated type

```
Block blockStar = new Block("star", "yellow");
Block polyStar = new Star();
Star trueStar = new Star();
```

- Let's have a look at each of these objects

BlockStar

```
Block blockStar = new Block("star", "yellow");
```

- Declared type: Block
  - object can be included in data structure that includes blocks (e.g. Block[])
  - object can only use methods declared in the Block class
- Instantiated type: Block
  - object uses the methods as they are written in Block class

PolyStar

```
Block polyStar = new Star();
```

- Most typical case in polymorphism
- Declared type: Block
  - object can be included in data structure that includes blocks (e.g. Block[])
  - object can only use methods declared in the Block class
- Instantiated type: Star
  - object uses the methods as they are written in Star class (fallback Block)

TrueStar

```
Star trueStar = new Star();
```

- Declared type: Star
  - object can be included in data structure that includes blocks (e.g. Block[])
  - object can only use methods declared in the Star class
- Instantiated type: Star
  - object uses the methods as they are written in Star class

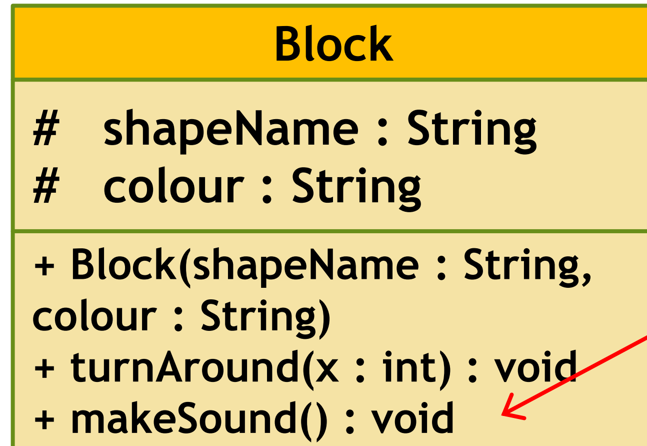
Methods and polymorphism

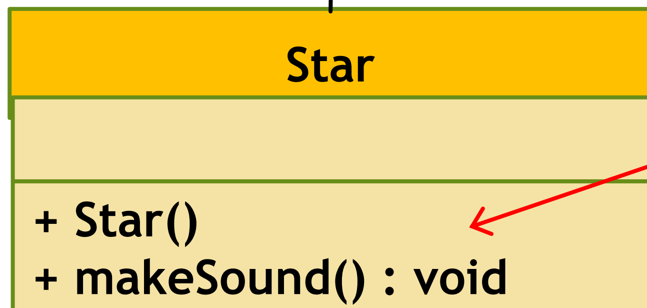
```
Block blockStar = new Block("star", "yellow");
Block polyStar = new Star();
Star trueStar = new Star();
```

- Methods specific to the Star class, non-existent in Block
  - Usable only by trueStar
- Methods existent in Block, but rewritten in Star
  - Usable by polyStar and trueStar
- Methods existent in Block, inherited as-is in Star
  - Usable by all three

How can we make polyStar able to use the method makeSound() ?

Method 1: Add "artificial" method






```
public class Block{
    ...
    public void makeSound(){
        // Leave method empty
    }
}
```

Indicates (in code) the fact that this code replaces code from superclass

```
public class Star extends Block{
    ...
    @Override
    public void makeSound(){
        System.out.println("Cling cling!");
    }
}
```

Abstract classes, abstract methods

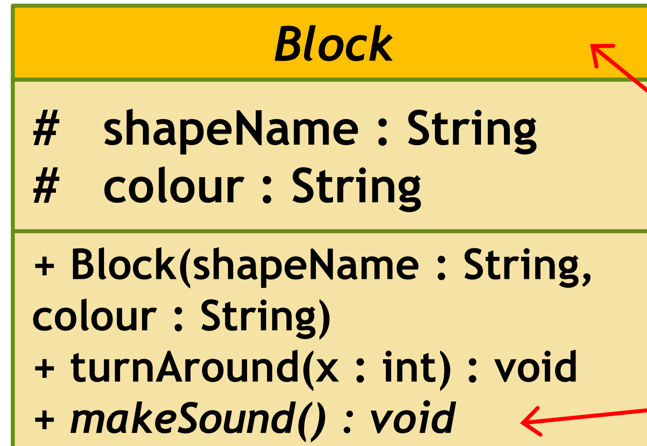
Abstract classes

- An abstract class is **non-instantiable**
  - That is, a class for which we cannot directly create objects

What is the use of such classes ?

- A subclass can inherit from an abstract superclass !
  - The abstract superclass can contain attributes and methods
    - Including a constructor!
    - Some methods concrete, others, abstract
    - Abstract method: **just the signature, followed by ;** (no details)
  - Concrete subclasses must detail all the abstract methods of the superclass

Method 2: polyStar using makeSound()




```
public abstract class Block{
    ...
    public abstract void makeSound();
}
```

keyword: abstract

Italics => abstract method/class

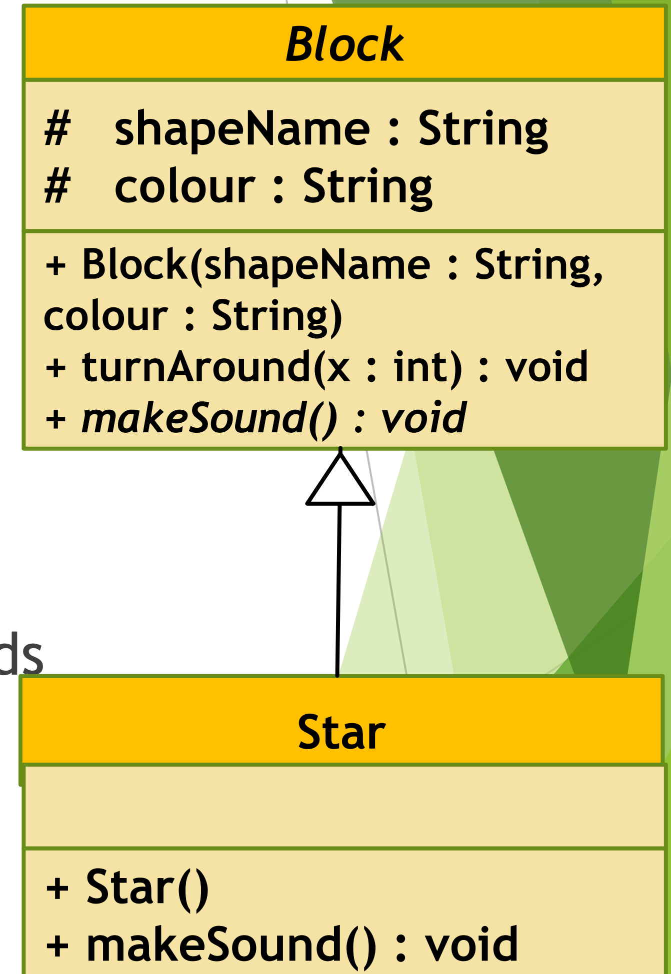
```
public class Star extends Block{
    ...
    @Override
    public void makeSound(){
        System.out.println("Cling cling!");
    }
}
```

Polymorphism & abstract methods

- Using abstract classes:
  - Objects can have Block as declared type…
  - … but not instantiated type

```
✗ Block blockStar = new Block("star", "yellow");
✓ Block polyStar = new Star();
✓ Star trueStar = new Star();
```

- Abstract classes allowed to contain only concrete methods
  - A class is abstract if it should never be instantiated as-is



Any questions ?