R2.01 : Object-oriented development (OOD)

<u>Coordinator : Isabelle Blasquez</u>
My name: Cristina Onete
cristina.onete@gmail.com
Slides : https://www.onete.net/teaching.html

Interfaces

Interfaces in Java

- **Inheritance** highlights similarities in what two types of objects **are like**:
  - A wolf, a human, and a bear are all mammals
- **Interfaces** highlight similarities in how things **behave**:
  - Instruments and games are both **playable**
  - We can **buy** a wide variety of objects
- Our goal: buy things that can be bought, play things that can be played
- ... irrespective of their classes

Intro to interfaces

- A Java interface is a set of characteristics (behaviours)
  - Described as abstract methods

- Classes that behave according to an interface **implement** it
  - And must (if concrete) detail all the interface's abstract methods

- Note: **an interface is not a class**!

Example

- Take a class **Person**
  - Persons have a name and a budget
- We also include these classes: **Burger**, **Instrument**, **Piano**, **VideoGame**
- We want persons to :
  - buy all these things
  - play instruments and video games
  - tune **a piano**

- Interfaces group classes depending on their functionalities:
  - Things that are ***buyable* :** Burger, Instrument, Piano, VideoGame
  - Things that are ***playable*** : Instrument, Piano, VideoGame
  - Things that are ***tunable*** : Piano

Example : our three interfaces

```
interface Buyable {
      double getPrice();
}
```

```
interface Tunable {
      void tune();
}
```

```
interface Playable {
      void play(Person[] players);
}
```

keyword: interface

abstract methods

all methods by default public

Example: class implementing an interface

- A burger is buyable

Keyword: implements

```
public class Burger implements Buyable{
    private String type;
    private double price;

    public Burger(String type, double price){
        this.type = type;
        this.price = price;
    }

    @Override
    public double getPrice(){
        return this.price;
    }
}
```

```
interface Buyable {
    double getPrice();
}
```

Burger is a **concrete class**
It must detail the **double getPrice()** method in interface **Buyable**

Example: class implementing 2 interfaces

- A videoGame is buyable and playable

Comma between the 2 interfaces

Compulsory, as VideoGame implements Buyable

Compulsory as VideoGame implements Playable

```
public class VideoGame implements Buyable, Playable{
        private String name;
        private double price;

        public VideoGame(String name, double price){
                this.name = name;
                this.price = price;
        }
        @Override
        public double getPrice(){
                return this.price;
        }
        @ Override
        public double play(Person[] players){
                // code for this method
                // must end with return statement
        }
}
```

Example: abstract class Instrument

- Abstract class
- Can implement interfaces
- Must include all the methods in interfaces
  - Since it is abstract though, the method need not detail those methods
  - Can have both abstract and concrete methods
  - Not obliged to include any of the methods that will remain abstract from the interfaces

```
public abstract class Instrument implements Buyable,
Playable{
      private String type;
      private String brand;
      private double price;

      public Instrument(String type, String brand, double
price){
            this.type = type;
            this.brand = brand;
            this.price = price;
      }

      @Override
      public double getPrice(){
            return this.price;
      }

}
```

Example: inheritance and interfaces

- Pianos are instruments that are, in addition, tunable

extends precedes implements

Concrete class
Can detail/implement this method, inherited from class Instrument

Piano implements Tunable

```
public class Piano extends Instrument implements Tunable {
    private int numberOfPedals;

    // ... constructor

    @Overide
    public double play(Person[] players){
        // code of method play
    }

    @Override
    public double tune(){
        // code of method tune
    }
}
```

Classes and interfaces

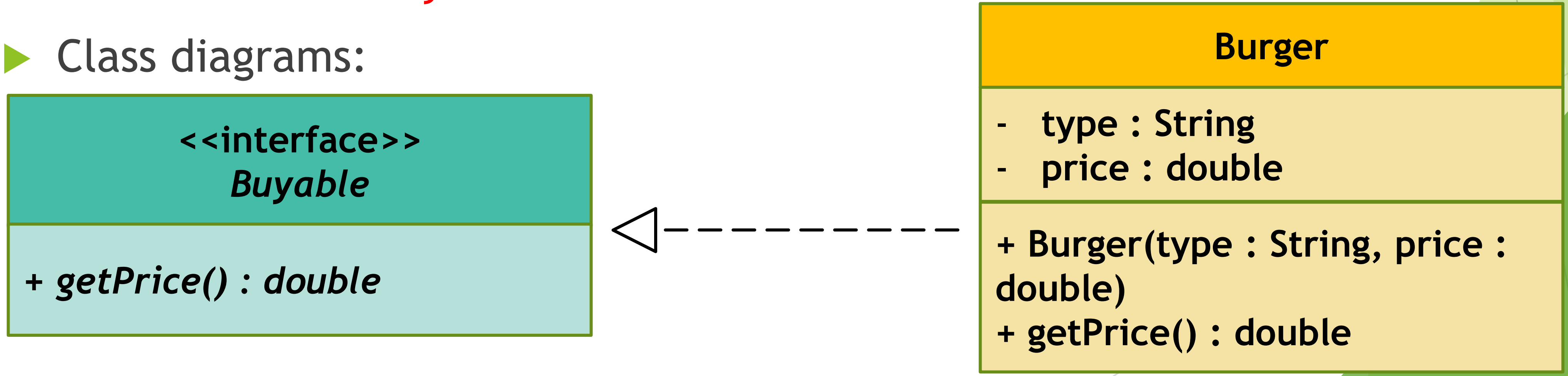
- A concrete class implementing an interface details **all its methods**
- An interface can be implemented by multiple classes
  - A class can also implement multiple interfaces

    ```
    class <ClassName> implements <Interface1>, <Interface2>
    ```

  - A class can inherit from at most one class & implement multiple interfaces

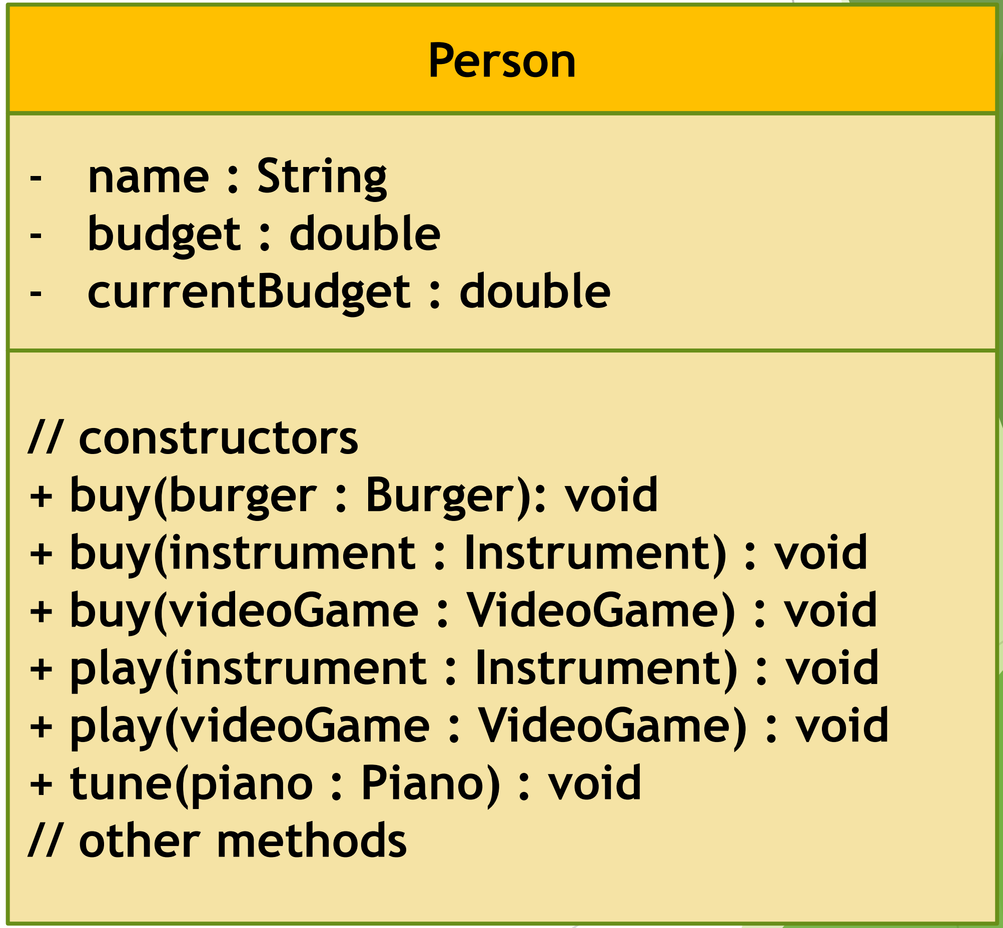
    ```
    class <ClassName> extends <Superclass> implements
    <Interface1>,<Interface2>
    ```

- Class diagrams:



Why interfaces are useful

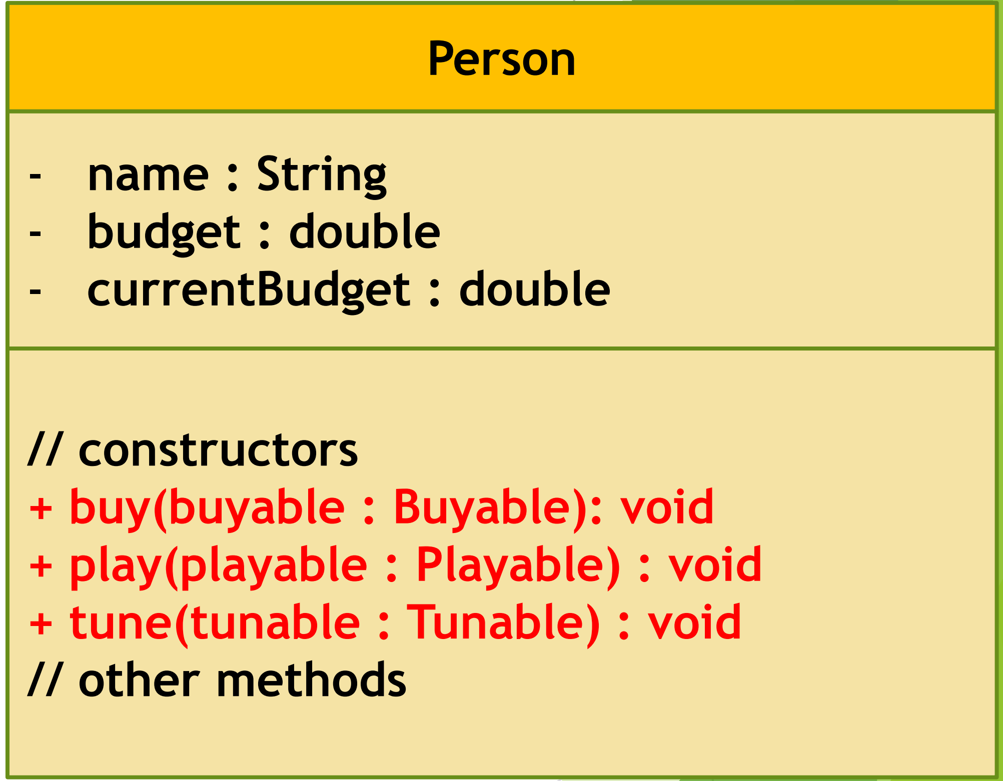
- Polymorphism can be used with interfaces
- A person must be able to buy: burgers, instruments, pianos, video games
- Without polymorphism: multiple methods:
  - `void buy(Burger burger)`
  - `void buy(Instrument instrument)`
  - `void buy(VideoGame videoGame)`
- More methods needed for playing (an instrument, a video game) or tuning (a piano)



```
Person

- name : String
- budget : double
- currentBudget : double

// constructors
+ buy(burger : Burger): void
+ buy(instrument : Instrument) : void
+ buy(videoGame : VideoGame) : void
+ play(instrument : Instrument) : void
+ play(videoGame : VideoGame) : void
+ tune(piano : Piano) : void
// other methods
```

Using interfaces

- A person must be able to buy: burgers, instruments, pianos, video games
- Let's use our interfaces: Buyable, Tunable, Playable
  - Single method `void buy(Buyable)` includes all buyables: a burger, an instrument, a video game
  - We can be sure that the right classes **contain the right methods** (`tune` for tunables, `getPrice` for buyables...)
- Same for `void play(Playable), void tune(Tunable)`



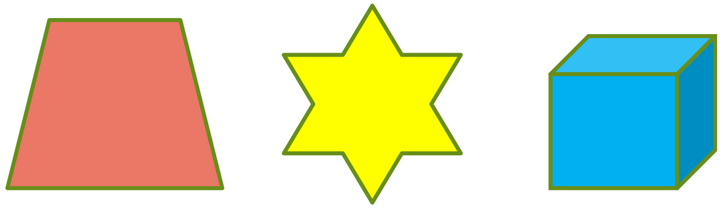
```
Person

-  name : String
-  budget : double
-  currentBudget : double

// constructors
+ buy(buyable : Buyable): void
+ play(playable : Playable) : void
+ tune(tunable : Tunable) : void
// other methods
```

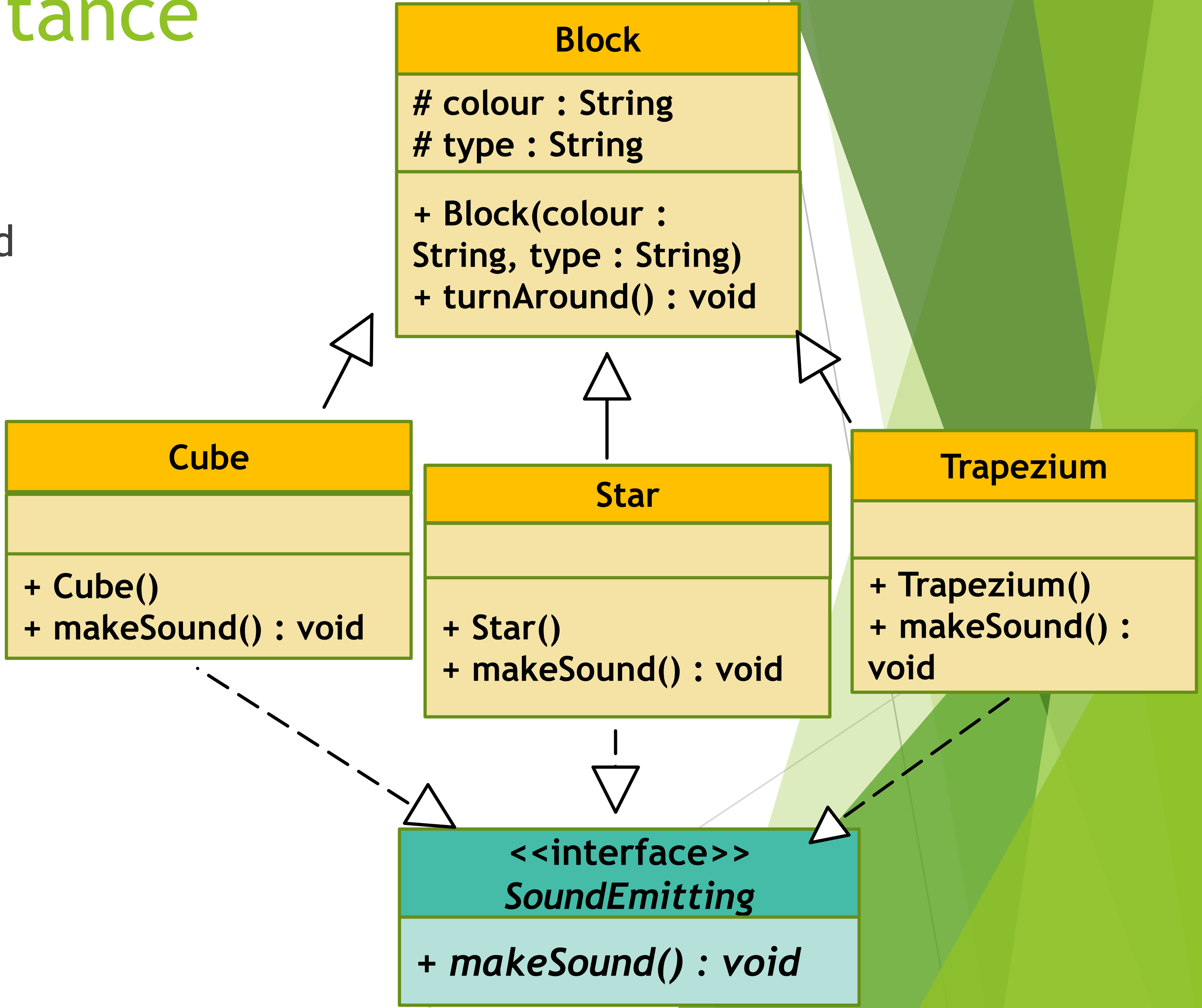
Interface vs. inheritance

Inheritance	Interface
• = "is a type of"	• = "works like a"
• Can inherit from **at most 1 superclass**	• Can implement **many interfaces**
• Keyword **`extends`**	• Keyword **`implements`**
• May contain concrete variables/methods	• Does not contain variables • Only abstract methods
• A subclass **may** redefine superclass methods (use of mention **@Override**)	• Concrete class **must** detail all the method of interfaces it implements

Interfaces and inheritance

- Recall our block example:
  - Each block makes different sound
  - Each block can be turned around





Collections

Beyond arrays

- Seen so far:
  - homogeneous arrays: variables instantiated as the same type
  - heterogeneous arrays: we use polymorphism

- Arrays can be very useful!
- ... however, they also present some disadvantages:

It is compulsory to declare the length of an array

Massively overestimating the length is costly

Once defined, the length cannot be changed

To add/remove elements we must indicate their index

Let's see an example!

- Take a group of users whose exact number is not known
  - ... and for which the number of users is hard to estimate
  - Say the number of users of a given social network

- We want to be able to:
  - **Store** in a single object all the objects representing users
  - **Add** and **remove** users at will
  - **Dynamically modify** the **size** of the container

- Idea: We could design a class NetUser which models users

Using an array of NetUsers

- First question: what would be the length of the array ?
  - Say 7 billion ? (all the users in the world)
  - But: even Facebook only has about 2.1 billion users...
  - That being said, the number of people using social networks keeps increasing every day

- Adding a user in the array:
  - Find the first open position and add the user there
- Changing the length of the array :
  - Declare and instantiate a new array of the updated length
  - Copy the non-null elements of the original array in the new one

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

No need to specify length

The type of the elements contained in userList
Beware: we use <> instead of []!

A call to the constructor of ArrayList
-- hence the parentheses

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
 ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
```

The size of the list becomes 1

This is a new object of type NetUser
(we tacitly assume the existence of the corresponding constructor)

The method add(Element) adds a new object into our ArrayList
The type of the added element must be the same as the type collected in the list

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add( 1, new NetUser("Jean Dupont"));
```

The methode add(index, Element) adds a new element at the given index

All other elements are shifted to the right

The size of the ArrayList increases by 1

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
 ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add(1, new NetUser("Jean Dupont"));
```

- Removing an element (a user):

```
userList.remove(user);
```

A previously-instantiated object of type NetUser

A better alternative: ArrayList

- An ArrayList is a collection of items of a single type:

```
 ArrayList<NetUser> userList = new ArrayList<NetUser>();
```

- Adding an element (a user):

```
userList.add(new NetUser("Jean Dupont"));
userList.add(1, new NetUser("Jean Dupont"));
```

- Removing an element (a user):

```
userList.remove(user);
```

- Preprogrammed methods to also find elements, clone the list, return an element situated at a given position…

Collections in general

- An ArrayList is just one example of a *Collection*
- *Java.util.Collection* is an interface in Java
  - ...which allows us to "collect" elements within a single structure
  - The elements of a collection MUST be objects
    - We cannot collect variables of primitive types
- Like any other interface, `Collection` only contains abstract methods
  - These are specified by the classes implementing the interface

The Java collection framework

Collections

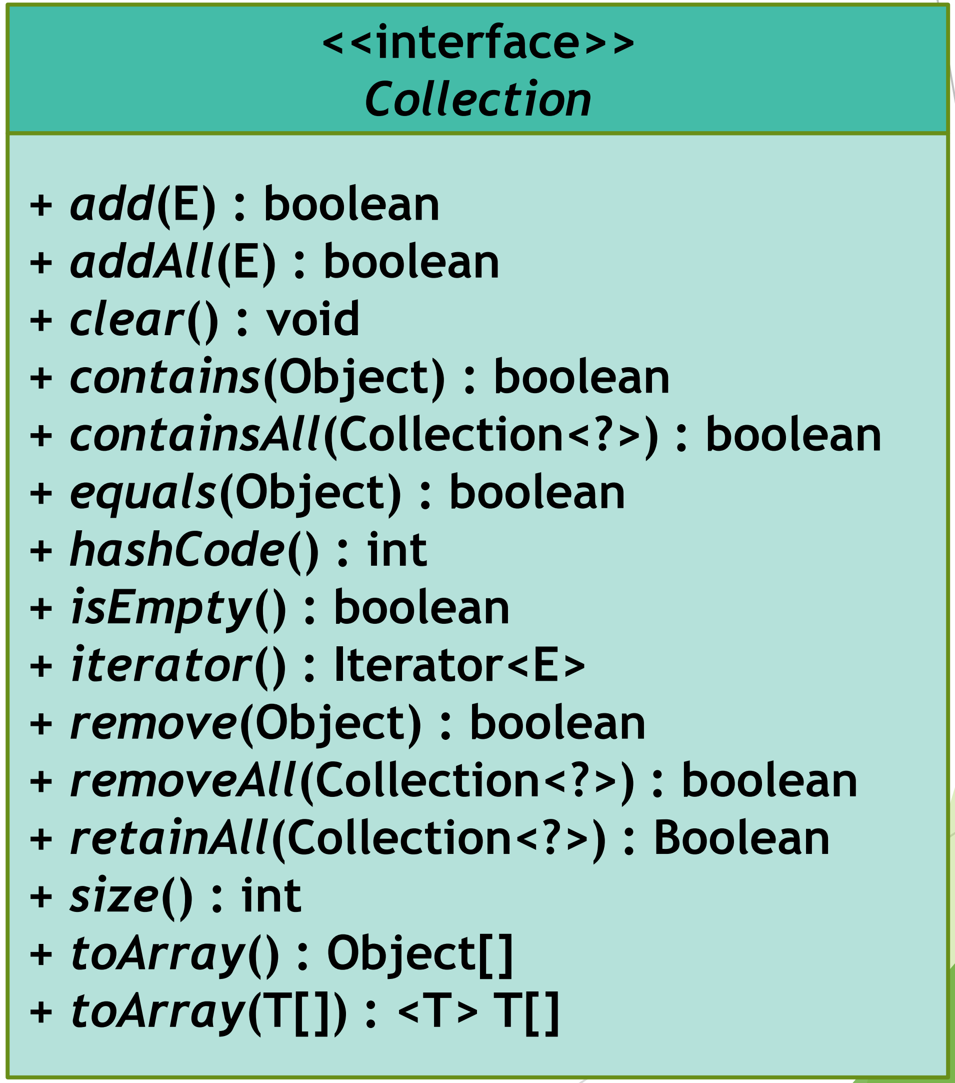
- The Java Collection Framework is a programming framework
  - It contains a hierarchy of **interfaces** for manipulating collections
  - Abstract methods in interfaces implemented concretely for various collections

- Why use collections:
  - Collections have variable sizes
  - Data structures and methods already provided in Java
  - A more efficient manipulation of the data
  - Interoperability

The generic structure of Collection

- The `Collection<E>` interface extends `Iterable<E>`

  In its turn `Iterable<E>` is useful in allowing us to generate iterators over the collection

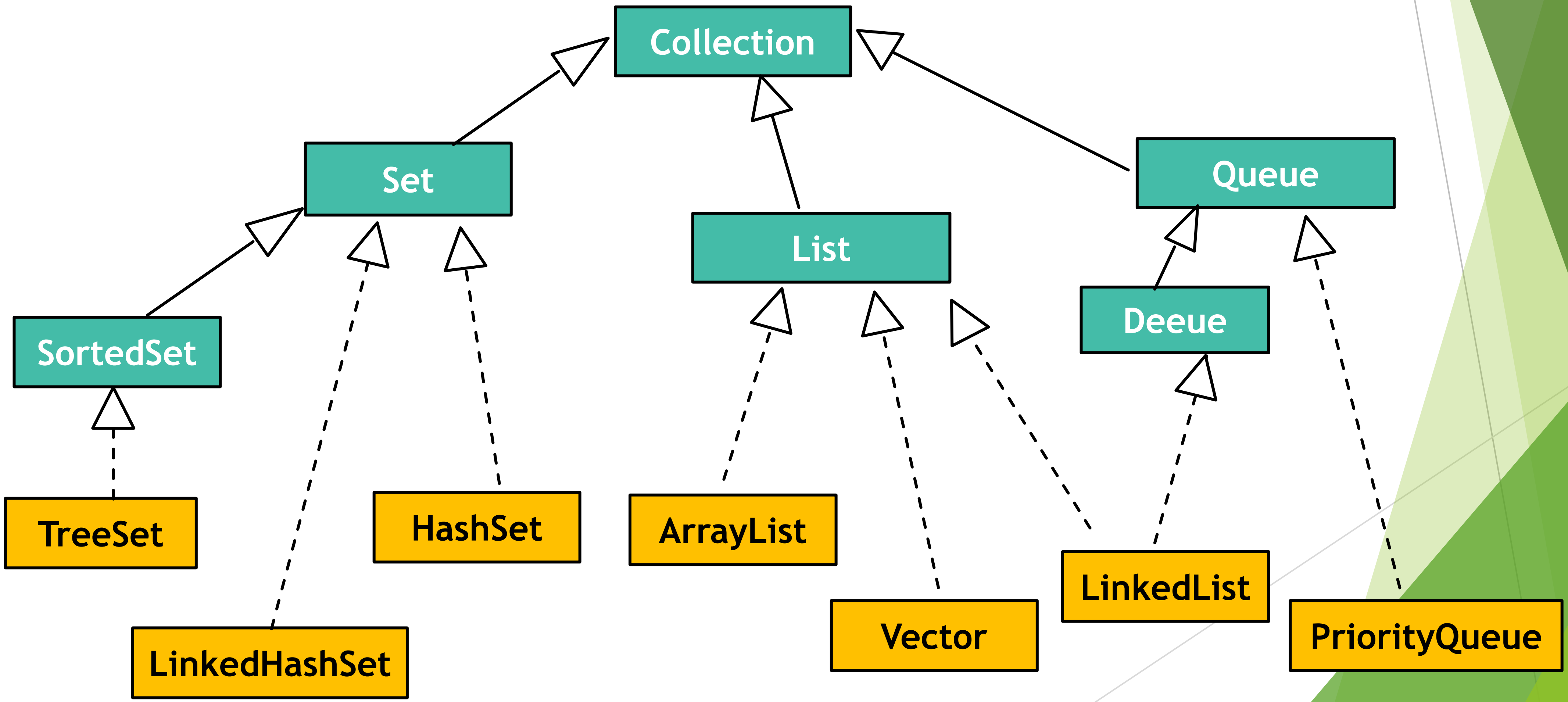


<<interface>>
Collection

+ *add*(E) : boolean
+ *addAll*(E) : boolean
+ *clear*() : void
+ *contains*(Object) : boolean
+ *containsAll*(Collection<?>) : boolean
+ *equals*(Object) : boolean
+ *hashCode*() : int
+ *isEmpty*() : boolean
+ *iterator*() : Iterator<E>
+ *remove*(Object) : boolean
+ *removeAll*(Collection<?>) : boolean
+ *retainAll*(Collection<?>) : Boolean
+ *size*() : int
+ *toArray*() : Object[]
+ *toArray*(T[]) : <T> T[]

The Java Collection Framework hierarchy

- Two main interfaces: Collection and Map

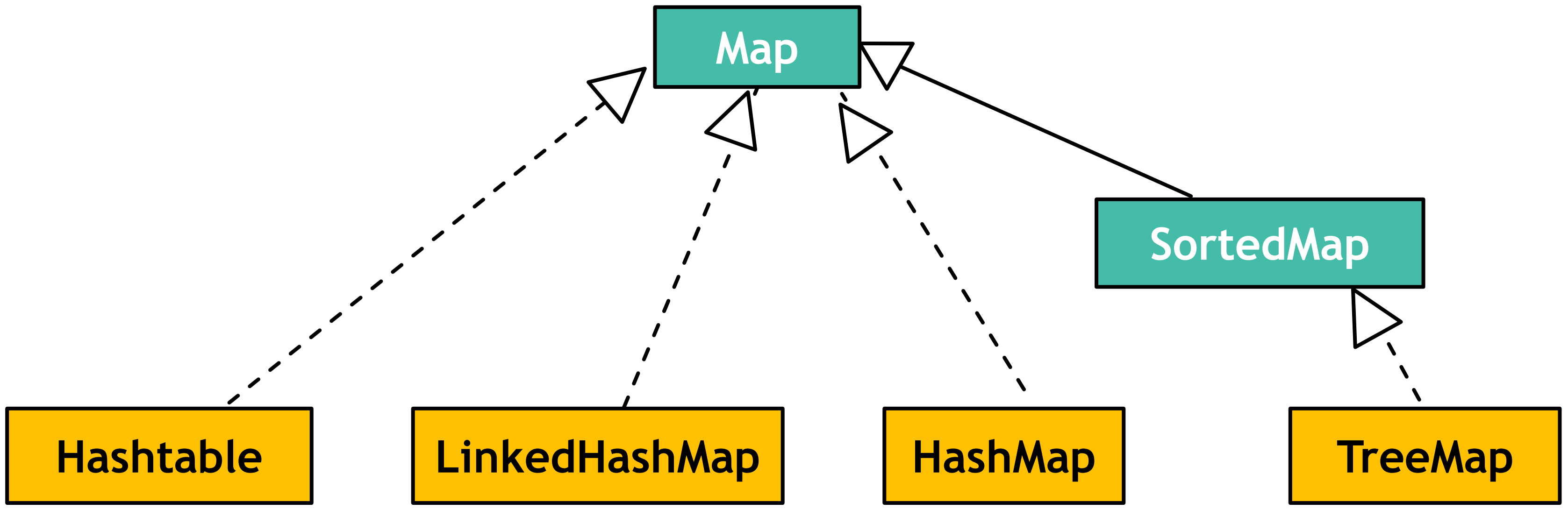


The Collection interface

- A generic interface for collections of objects

- Different collections implement different subinterfaces:
  - **Set**: a collection without repeated elements
  - **SortedSet**: a set whose elements are ordered increasingly
  - **List**: a collection allowing for duplication
    the elements are indexed
  - **Queue** : a collection that stores elements awaiting processing
  - **Deque** (double-ended queue) : a queue allowing to insert and process
    elements at both ends of the queue

The Map interface



- The Map et SortedMap interfaces:
  - Map: associates values to keys
  - SortedMap : a map that is sorted by key values

Some collections

Sets

- An unindexed collection with no repeated elements
  - We can never access an element by using its "position"
- Set vs. array:
  - Sets can have an arbitrary, modifiable size
  - Elements are easier to access in arrays, by using their indexes
  - Sets have inbuilt methods for simple element manipulation: adding/removing elements, checking if an element is in the set...

Using a set

- A collection with no repetitions
- An interface with specific methods of interface Collection, with restrictions on repetitions
- Classes implementing Set:
  - HashSet: the elements are stored in a hash table (yielding constant-time operations)
  - TreeSet: the elements are stored in order in a tree
  - LinkedHashSet: the elements are stored in order given their moment of insertion

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");
        mySet.add("John"); // not added (repetition)
        System.out.println(mySet); // prints {John,
Gabi} or {Gabi, John}
        System.out.println(mySet.size());   //
prints 2

}
```

Iterators and printing

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");

        Iterator<String> iterator =
mySet.iterator();
        while (iterator.hasNext()){
            String currentItem = iterator.next();
            System.out.println(currentItem);
            if (currentItem.equals("John"))
                System.out.println("It's John");
        }
    }
}
```

Iterators are useful for iterating through an unordered collection

The iterator is specific to `mySet`

The method `hasNext()` checks if iterator still has room to iterate

The `next()` method returns the object the iterator will encounter next

Compares elements

Iterators and printing

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");

        Iterator<String> iterator = mySet.iterator();
        while (iterator.hasNext()){
            String currentItem = iterator.next();
            System.out.println(currentItem);
            if (currentItem.equals("John"))
                System.out.println("It's John");
        }
    }
}
```

We can view the set as a text, in which the words are set one space apart

```
Gabi John
```

An iterator is like a cursor in the text

Evtery time we call next() :

- the cursor moves one position to the right
- the object between its former and current positions becomes currentItem

Other operations with sets

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class LearnSets{
    public static void main(String[] args){
        final Set<String> mySet = new
HashSet<String>();
        mySet.add("John");
        mySet.add("Gabi");
        System.out.println(mySet.size());   //
prints 2

        mySet.remove("Gabi");
        System.out.println(mySet); // prints John
        mySet.add("Alice");
        if (mySet.contains("Gabi"))
            System.out.println("Gabi is still
here.");
    }
}
```

removes an element

checks if the set contains a given element -- no need to actually iterate through it!

Using lists

- Lists are indexed
- Repetitions are allowed
- Additional methods:
  - To handle elements whose index we know
  - To partition a list in sublists
  - To sort, mix, invert, copy or find elements
- Types of lists:
  - ArrayList: a resizable array
  - LinkedList : a list that is manipulable at both ends

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class LearnLists{
    public static void main(String[] args){
        final List<Integer> myList = new
ArrayList<Integer>();
        myList.add(Integer.valueOf(10));
        myList.add(Integer.valueOf(5));
        myList.add(Integer.valueOf(8));
        myList.add(Integer.valueOf(10)); //added
        System.out.println(myList); // prints
{10,5,8,10}
        System.out.println(myList.get(2));   //
prints 8, as indexing starts at 0
}
```

Integer: a class that encapsulates an int into an object

Remember: collections only contain objects!

More ways to handle lists

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class LearnLists{
    public static void main(String[] args){
        final List<Integer> myList = new
ArrayList<Integer>();
        myList.add(Integer.valueOf(10));
         myList.add(Integer.valueOf(5));
         myList.add(Integer.valueOf(8));
         myList.add(Integer.valueOf(10));

        System.out.println(myList.indexOf(10));
// prints 0
        System.out.println(myList.lastIndexOf(10));
// prints 3
        Collection.sort(myList);
        System.out.println(myList); //prints
{5,8,10,10}
}
```

Part of the Java Collection Framework
Includes many abstract methods, including sorting

Finds the first index of a given element

Finds the last index of a given element

A method that works on all lists with sortable elements
Including lists on Strings and Integers

Using a Map

- A map consists of pairs (key, value) such that keys are unique
  - Example of keys: student numbers
  - The keys are used to index the data
- Methods allowing us to manipulate elements by keys, or keys themselves
- Types of maps:
  - HashMap: keys/elements stored in a hash table (more efficient)
  - TreeMap: elements stored in a tree (and ordered)
  - LinkedHashMap: tradeoff between HashMap (efficient) et TreeMap (ordered)

```
import java.util.HashMap;
import java.util.Map;

public class LearnMaps{
    public static void main(String[] args){
        final Map<String, String> 00Agents = new
HashMap<>();
        00Agents.put("002", "Bill Fairbanks");
        00Agents.put("006", "Alec Trevelyan");
        00Agents.put("007", "James Bond");
        00Agents.put("002", "Johnny English"); // not
added, duplicate key
        System.out.println(00Agents); // prints {002=Bill
Fairbanks, 006=Alec Trevelyan, 007=James Bond}
}
```

Other operations with Maps

```
import java.util.HashMap;
import java.util.Map;

public class LearnMaps{
    public static void main(String[] args){
        final Map<String, String> 00Agents = new
HashMap<>();
        00Agents.put("002", "Bill Fairbanks");
        00Agents.put("006", "Alec Trevelyan");
        00Agents.put("007", "James Bond");
        System.out.println(00Agents.get("007"));
        System.out.println(00Agents.size());
        if (00Agents.containsKey("001")){
            System.out.println(00Agents.get("001"));
        }
        else{
            System.out.println("Vacant position.");
        }
        System.out.println(00Agents.keySet());
        System.out.println(00Agents.values());
}
```

Find the value associated to the key "007"

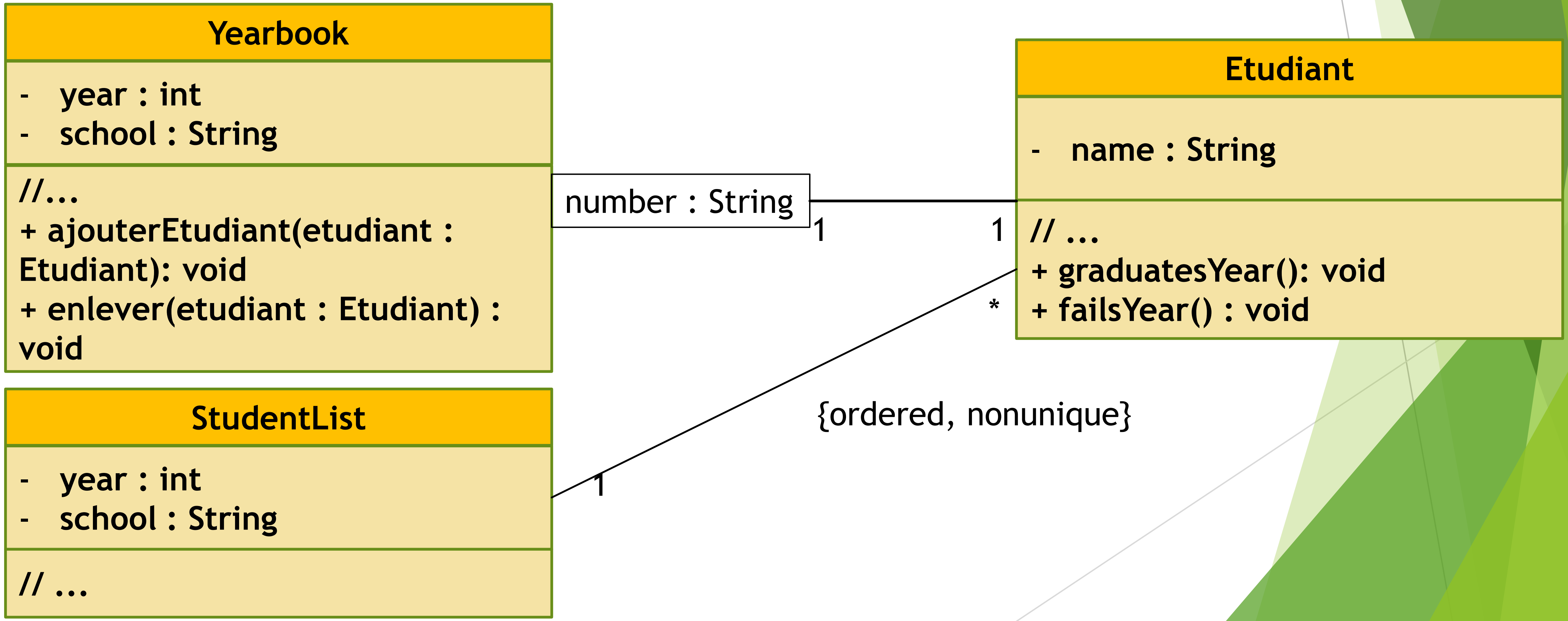
Searches through the keys
We can search values by using **contains()**

Prints a set of keys: {"002", "006", "007"}

Prints a set of values

Class diagrams

- Collections are generally indicated by means of associations
  - A less standard notation: use attributes directly


Yearbook
- year : int
- school : String
//...
+ ajouterEtudiant(etudiant : Etudiant): void
+ enlever(etudiant : Etudiant) : void
number : String
1
1
Etudiant
- name : String
// ...
+ graduatesYear(): void
+ failsYear() : void
*
{ordered, nonunique}
1
StudentList
- year : int
- school : String
// ...