School of Computer Science

Unit 7: Sophisticated behavior

Introduction to Programming

Academic year 2023-2024

Concepts

- Other control flow statements: **do-while** and **switch**
- The **? :** ternary operator
- Using **library classes**
- Generating **random numbers**
- More collections**: HashSet** and **HashMap**
- **Class** methods, variables and constants
- Writing **documentation**

Review …

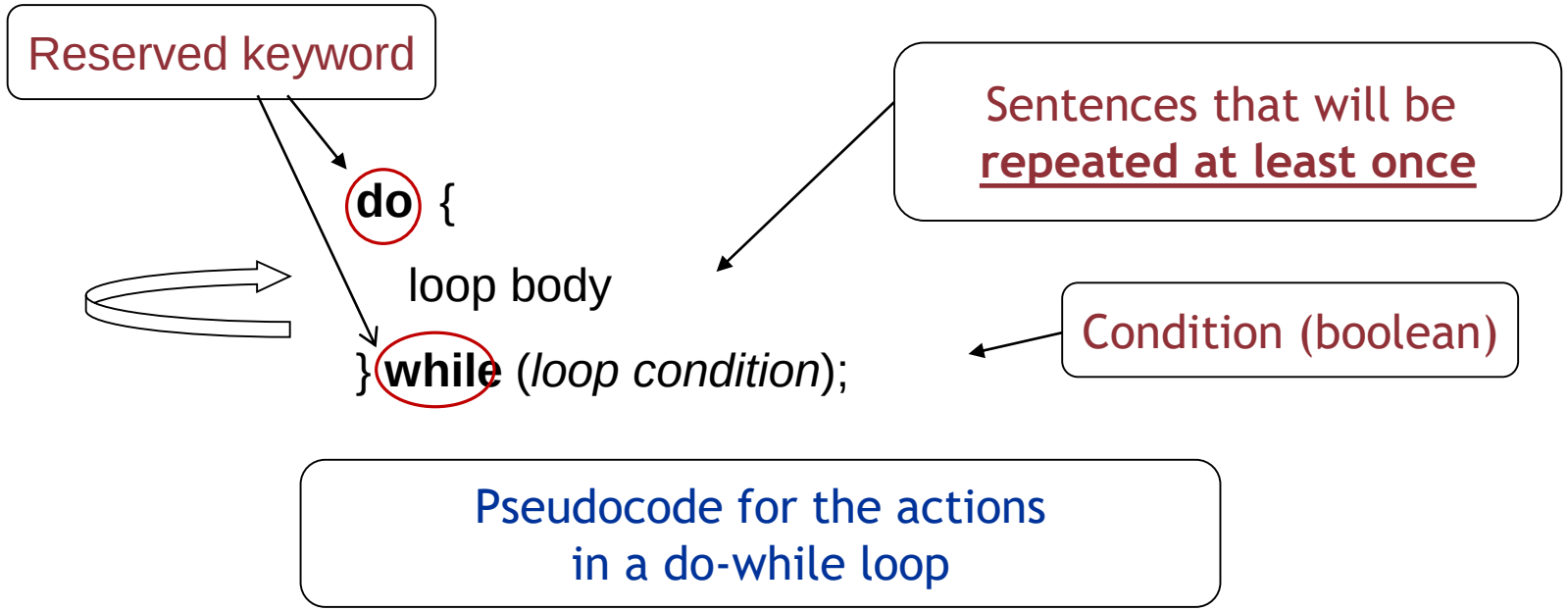
- **for-each loop**
  - It processes all elements in a collection.
  - It can be applied to both fixed-size and flexible-size collections.
- **for loop**
  - It allows us to process a collection either totally or partially.
  - It can be used with both fixed-size and flexible-size collections.
  - It can be used to repeat the execution of a block of sentences a given number of times. In that case it is used without a collection.

Review …

- **while loop**
  - It can process a collection either totally or partially.
  - It can be used with fixed-size and flexible-size collections.
  - It can be used to repeat the execution of a block of sentences. In that case it is used without a collection.

- **`Iterator` objects**
  - They allow us to process a collection either totally or partially.
  - Available for all kind of collections in the Java Library.
  - Usually employed with collections where index-based access is impossible or inefficient.

do-while loop



Execute the sentences inside the loop body **while** the loop **condition is true**

do-while example

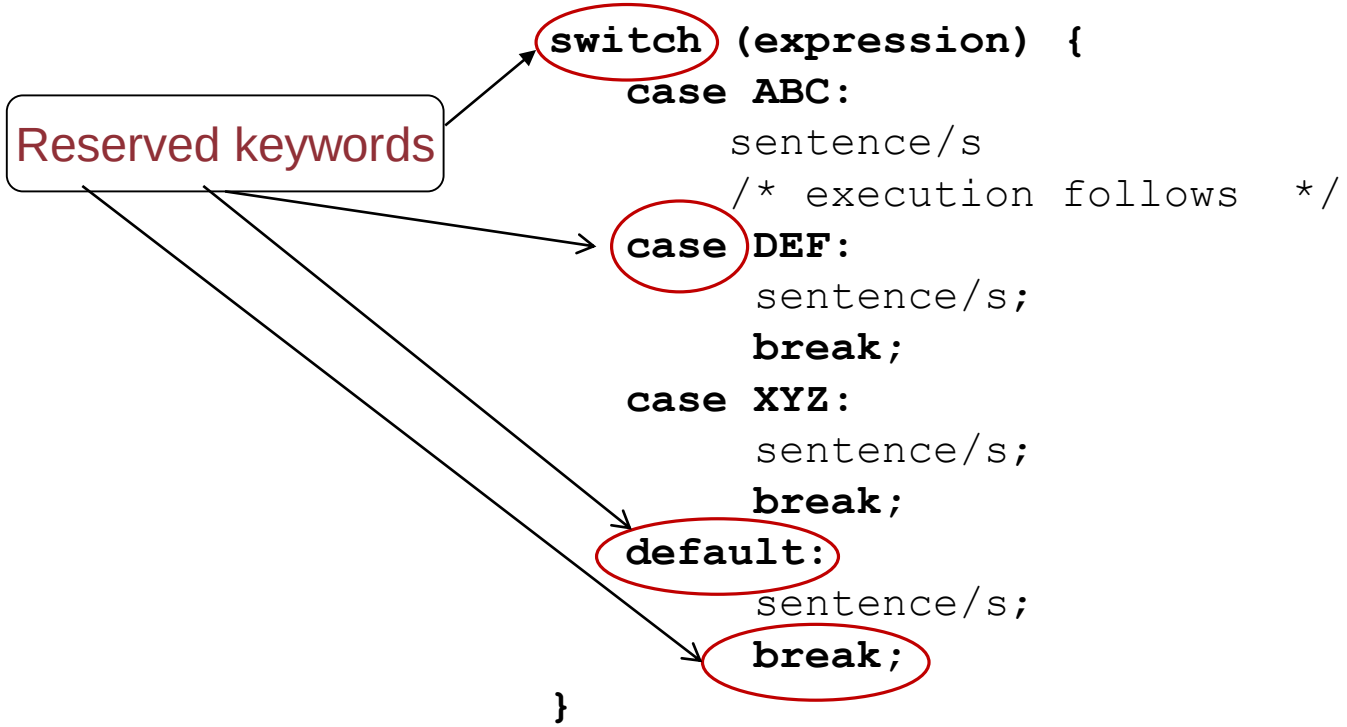
```
int number = 4557888;
int digits = 0;
do {
   number /=10;
   digits ++;
}
while (number > 0 );
```

do-while loop

- **do-while loop**
  - It can process a collection partially or totally.
  - It can be used both with fixed-size and flexible-size collections.
  - It can be used to repeat the execution of a block of sentences. In that case it is used without a collection.

switch-case statement

General form of the switch-case statement



Multialternative statement. The *break* sentence stops the execution and jumps to the sentence immediately after the *switch-case*.

switch-case example

```
public void miniCalculator(int a, int b, char op){
  System.out.print("Result is: ");
  switch ( op ) {
  case '+':
    System.out.println( a + b );
    break;
  case '-':
    System.out.println( a - b );
    break;
  case '*':
    System.out.println( a * b );
    break;
  case '/':
    System.out.println( a / b );
    break;
  default:   //default is optional
    System.out.println("error" );
    break;   //break HERE is optional
  }
}
```

switch-case example

```
public void changePosition(char orientation) {
  switch (orientation) {
      case 'n':case 'N':
        setPosY(getPosY()-1));
        break;
      case 'e':case 'E':
        setPosX(getPosX()+1);
        break;
      case 's':case 'S':
        setPosY(getPosY()+1);
        break;
      case 'w':case 'W':
        setPosX(getPosX()-1);
        break;
      default:
        setPosX(getPosX()+1);
        setPosY(getPosY()+1);
    }
}
```

Ternary operator ?:

Used to choose one of two possible values, depending on the result of the evaluation of a boolean expression. Its syntax is the following:

```
condition ? expressionIfTrue : expressionIfFalse
```

A)

```
if (value == 1){
   System.out.println("*");
} else{
   System.out.println(" ");
}
```

B)

```
String  whatToprint = value == 1 ? "*" : " ";
System.out.println(whatToprint);
```

C)

```
System.out.println(value == 1 ? "*" : " ");
```

Using library classes

- The Java class library contains lots of useful classes.

You must:

- Know some important classes by their name.
- Know how to locate other classes.

Important:

- You only need to know the interface, not the implementation.

Using library classes

- Library classes must be imported using the keyword ***import*** (except for classes within the package ***java.lang*** that are automatically imported).
  - The following classes belong to *java.lang*: **`Object`**, **`String`**, **`Exception`**, **`...`**
- Classes from this library can be used within any project we are implementing.

Packages and `import`

- Classes are organized in packages.
- You can import a single class:

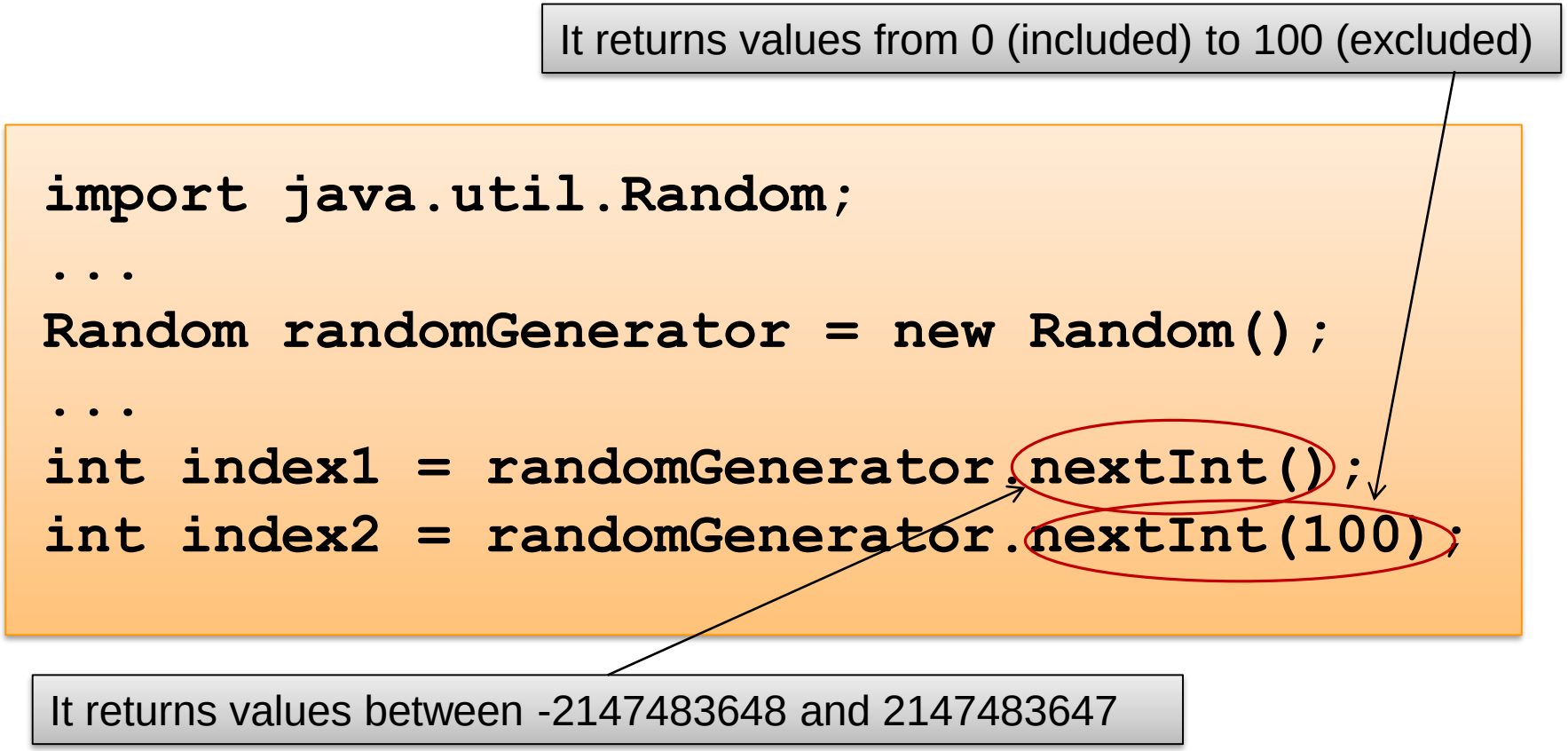
```
import java.util.ArrayList;
```

- Or you can import whole packages (all classes inside the package):

```
import java.util.*;
```

Using Random

- The `Random` class can be used to produce random numbers



Exercises

- Code a method called `throwDice` that returns a number between 1 and 6 (both included)
- Code a method `produceValue` with two parameters `min` and `max` that produces a random number between `min` and `max` (both included)
- Code a method `getAnswer` that produces randomly one of the following strings: `"yes"`, `"no"` or `"maybe"`.

Example

```
public void answer() {
    ArrayList<String> answers = new ArrayList<String>();
    fillAnswers();
    System.out.println(generateAnswer());
}

public String generateAnswer() {
    Random random = new Random();
    int index = random.nextInt(answers.size());
    return answers.get(index);
}

public void fillAnswers() {
   ...
}
```

Try to find the error …

```
import java.util.ArrayList;

class PersonalOrganizer {

  private ArrayList<String> notes;

  public PersonalOrganizer()  {
    ArrayList<String> notes = new ArrayList<String>();
  }

  public void addNote(String note) {
    notes.add(note);
  }
}
```

Sets

- A set is a collection that stores **each individual item at most once**. It does **not maintain any order.**
  - Iterators can return items in an order different from that in which they were added.
  - Adding an element more than once does not produce any effect.
- In an *ArrayList*, elements are ordered, they are accessed by means of indices and the same element can be stored more than once.

Using sets

```
import java.util.HashSet;

...
HashSet<String> mySet = new HashSet<String>();

mySet.add("one");
mySet.add("two");
mySet.add("three");

for(String element : mySet) {
    // do something with each element
}
```

Compare this with the ArrayList version

Splitting Strings

```
public HashSet<String> makeSet(String line) {

    String[] wordsArray = line.split(" ");
    HashSet<String> words = new HashSet<String>();

    for(String word : wordsArray) {
        words.add(word);
    }
    return words;
}
```

This method splits a String in different substrings returning them as an array of Strings.

Questions…

```
public HashSet<String> makeSet (String line) {

    String[] wordsArray = line.split(" ");
    HashSet<String> words = new HashSet<String>();

    for(String word : wordsArray) {
       words.add(word);
    }
    return words;
}
```

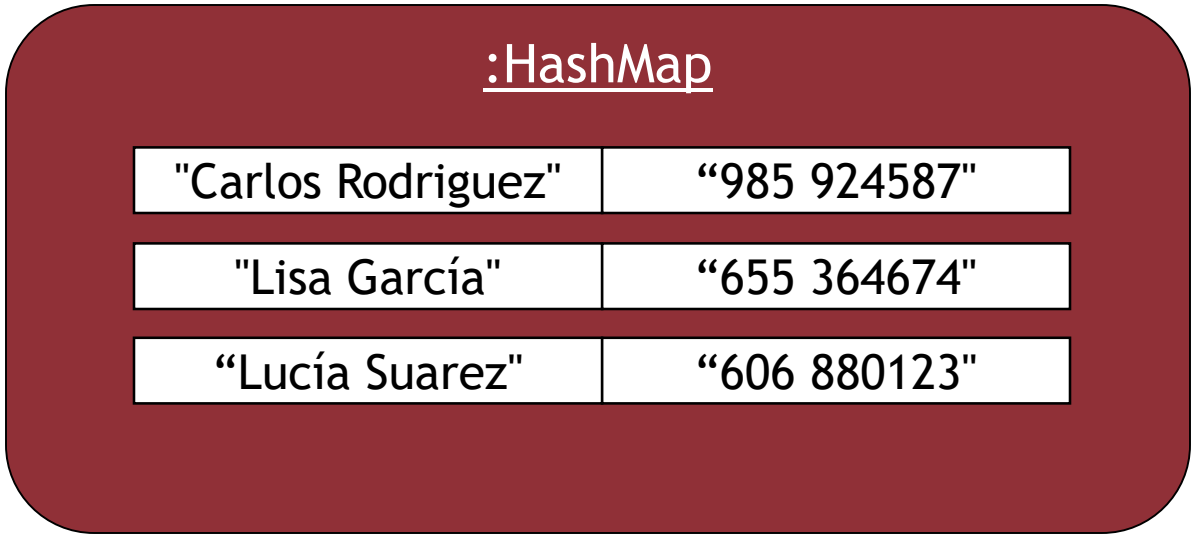
What happens if there are more than one blank between words?
And if we use punctuation symbols?

Maps

- A map is a **collection of pairs** (key, value). Values are looked for using keys.
  - Adding an element (a given pair) twice, does not produce any effect.
- In an *ArrayList,* elements are ordered, they are accessed by means of indices and the same element can be stored more than once.

Using a HashMap

- A map with Strings as both keys and values.



Using a HashMap

```
HashMap <String, String> contacts = new HashMap<String,
String>();

contacts.put("Carlos Rodriguez", "999 924587");
contacts.put("Lisa García", "666 364674");
contacts.put("Lucía Suarez", "666 880123");

String phoneNumber = contacts.get("Lisa García");
System.out.println(phoneNumber);
```

Questions…

- What happens if you try to add an entry with an already used key?
- What happens when you try to add an entry with an already used value?
- How can you verify if a map already has a given key?
- How can I know how many entries are stored in a map?

Using a HashMap

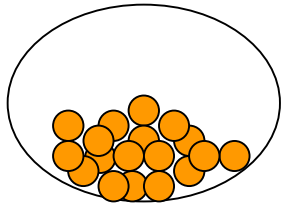
Processing the contents of the collection

```
HashMap <String, String> contacts = new HashMap<String, String>();

contacts.put(" Carlos Rodriguez ", " 999 924587 ");
contacts.put(" Lisa García ", " 666 364674 ");
contacts.put(" Lucía Suarez ", " 666 880123 ");
contacts.put(" Lucía Suarez ", " 666 111111 ");

// Version 1
Iterator<Map.Entry<String,String>> it = contacts.entrySet().iterator();
while (it.hasNext()) {
      Map.Entry<String,String> elem = it.next();
      System.out.println(elem.getKey() + " " + elem.getValue());
}
// Version 2
for(Map.Entry<String,String> elem : contacts.entrySet())
        System.out.println(elem.getKey() + " " + elem.getValue());
```

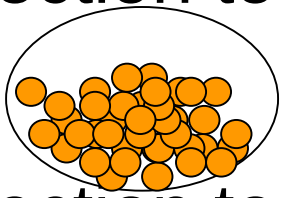
Questions…

For the Spanish Christmas Lottery…

- Which is the best collection to store the ticket numbers?



- Which is the best collection to store the prizes?



- Which is the best collection to store the results of the drawing?

Access modifiers

public *vs* private

- Public attributes, constructors and methods are accessible from both the own class and from other classes.
- Attributes **should not** be public.
- Private attributes are accessible just to the class where they are defined.
- Only those methods that are to be called from other classes should be public.

Access modifiers

Without reserved keyword

- If you do not specify neither public nor private the access is “package” and any class inside that package has access to the member (attribute, method or constructor).

This is the default access!

Information hiding

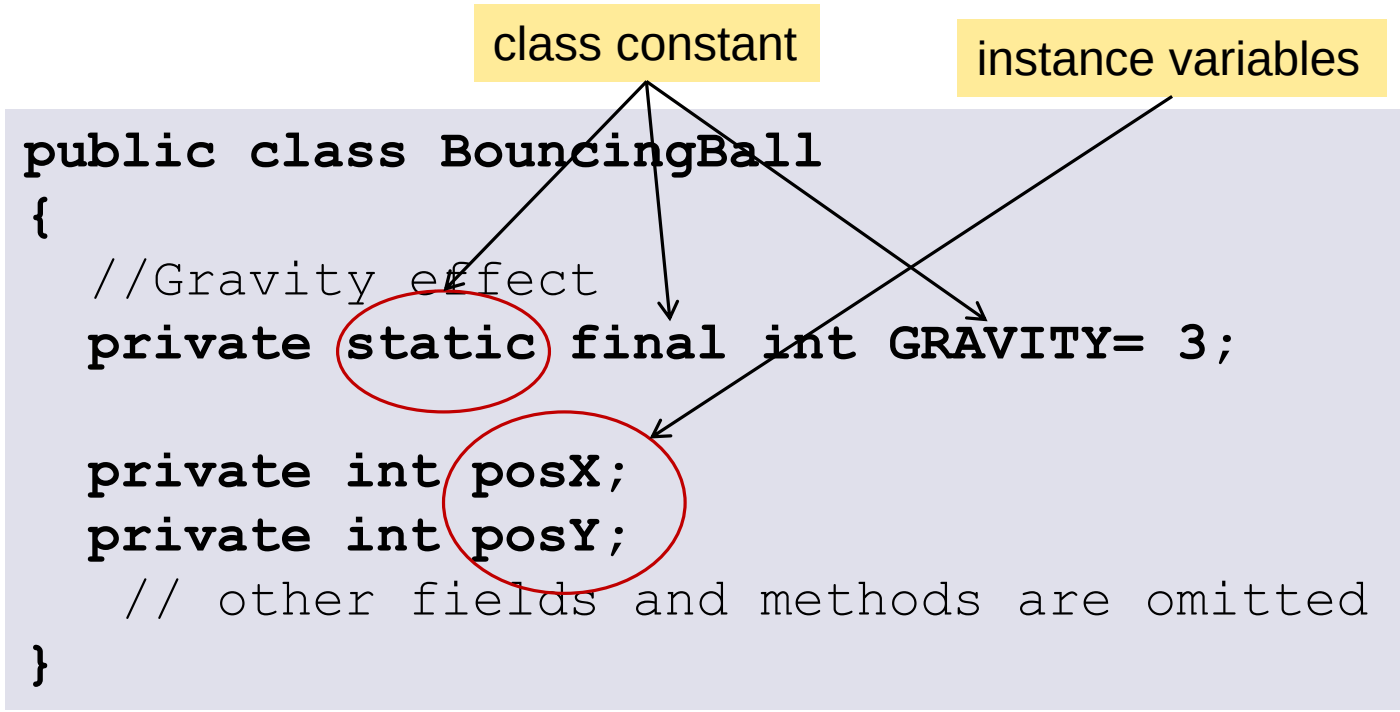
- The inner implementation details of a given class should be hidden to other classes.

Abstraction and modularization

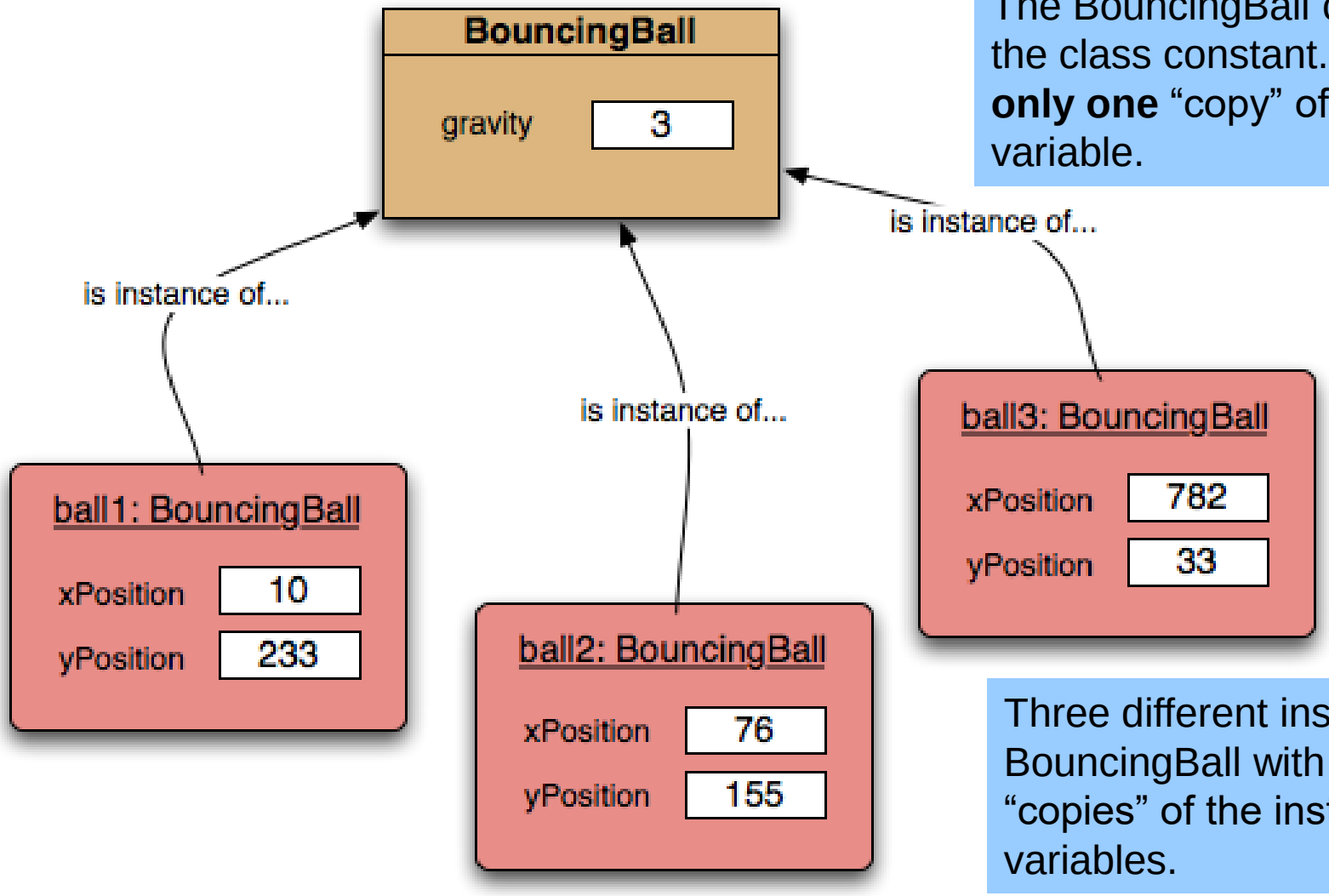
"If we needed to know all the inner details of all the classes we need to use, we would be unable to build large systems".

Class variables and constants

- **Class variables and constants** are fields (attributes) that are stored **not** in an object but in the class where they are defined.
- **Instance variables** are stored in each object.



Class variables and constants



The BouncingBall class stores the class constant. There is **only one** "copy" of that variable.

Three different instances of BouncingBall with their own "copies" of the instance variables.

Class constants

```
private static final int GRAVITY = 3;
```

`private`: access modifier

`static`: class variable

`final`: constant

Example

```
public class StreetLight {
  public static final boolean ON = true;
  public static final boolean OFF = false;

  private boolean status = OFF;

  public void setStatus(boolean status) {
    this.status = status;
  }
  ...
}
```

```
StreetLight light = new StreetLight();
light.setStatus(StreetLight.ON);
```

Class methods (static methods)

- **Static methods** always produce the same outcome without regards to the object.
  - They can be executed **without creating** any object (they are "controlled" by the class).
  - They have **access only to static attributes**.

Example The Math class

```
Math.PI //class constant
Math.sqrt() //class method
```

Example

Class variables

Instance variable

```
public class TenEurosNote
{
    private static double rate = 1.09;
    private static String color = "red";
    private static double inDollarsValue = rate*10;
    private String serialNumber;

  public TenEurosNote(String serialNumber) {
        setSerialNumber(serialNumber);
    }

  // Static method to set the exchange rate with $
  public static void setExchangeRate(double rate){
      setRate(rate);
      setInDollarsValue(10*getRate());
    }
}
```

Writting class documentation

- We must document our classes in the same way library classes are documented.
- That way, other people could use our classes without knowing their implementation.
- So, our classes become … library classes!

Elements in documentation

Documentation for a class must include:

- The class name.
- A comment describing the general purpose of the class and its features.
- A version number.
- The author(s) name(s).
- Documentation for each constructor and method.

Javadoc

Class comment

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (30.Mar.2006)
 */
```

Elements in documentation

Documentation for methods and constructors must include:

- The method name.
- The return type (for methods).
- Name and type of parameters.
- A description of the general purpose of the method.
- A description for each parameter.
- A description for the value it returns (if any).

Javadoc

Method comment

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param  prompt  A prompt to print to screen.
 * @return A set of Strings, where each String is
 *         one of the words typed by the user
 */
public HashSet<String> getInput(String prompt)
{
    ...
}
```

Javadoc. Some tips

- **Methods** implement actions. So, verbs must be used to describe their function.

  Calculates the average age of the passengers

- **Try to avoid** references to internal the representation, names, attributes or values of the class.

  Calculates the average of the elements of the bidimensional array.

- When referring something of the object, use *this* instead of *the:*

  Gets the age of ***this*** *person*

Interface and implementation

- The interface of a class describes what the class is able to do and the way in which you can use it without disclosing the implementation.

The documentation for the Java class library provides:

- The class name.
- A general description of the class' purposes.
- The list of its constructors and methods.
- The parameters and return types for each constructor and method.
- A purpose description for each constructor and method.

The class interface

Field Summary

static java.lang.String	**ADULT_STATUS** A string representing the intermediate age interval.
static double	**ADULTHOOD_AGE** Age from which a person is considered an adult.
static java.lang.String	**CHILD_STATUS** A string representing the lowest age interval.

Constructor Summary

Person()
Creates a Person object with the following values for its properties
- name --> "Fernando"
- age --> 35.0
- telephone--> "985212121"
- weight --> 75.0

Person(double age)
Creates a Person object with the by default values for its properties but his age, which will take the value given as a parameter

Method Summary

int	**compareToAge**(Person other) Returns one of the following values: 0: if the age of the host and parameter objects is the same.
boolean	**compareToName**(Person other) Returns one of the following values: true: if the age of the host and parameter objects is the same.
double	**getAge**() Returns the current age of the person
double	**getCriticalAge**() Calculates one of the three following values:

Interface and implementation

- The whole code defining a class is called **implementation**.

*Documentation **does not** include:*

- Private fields (most of them).
- Private methods.
- Methods' bodies.

→ **The class implementation**

Interface and implementation

- The **interface** of a method consists of its signature and a comment.

The documentation of the Java class library provides:

- The access modifier (public, private, …)
- The return type for the method.
- The name of the method.
- A parameter list (it can be empty)

The method's interface

- It provides all the elements you need to know how to use it.

Method Detail

compareToAge

```
public int compareToAge(Person other)
```

Returns one of the following values:
0: if the age of the host and parameter objects is the same.
-1: if the age of the host object is less than the age of the parameter object.
1: other case.

Parameters:
`other` - A reference to another Person object to which compare the age
Returns:
One of the above values

---

compareToName

```
public boolean compareToName(Person other)
```

Returns one of the following values:
true: if the age of the host and parameter objects is the same.
false: other case.

Parameters:
`other` - A reference to another Person object to which compare the age
Returns:
One of the above values

---

getAge

```
public double getAge()
```

Returns the current age of the person

Returns:
The current age of the person

In short

- Java has large class libraries.
- You should be familiar with them.
- Documentation provides us what we need to know to use a class (interface).
- Implementation is not visible (information hiding).
- We have to document our own classes.