# Fundamentals of Programming 2

## Lecture 8

**Aurelia Power, TU Dublin – Blanchardstown Campus, 2019**

*** Notes based on the Java Oracle Tutorials (2018), Deitel & Deitel (2015), and Horstman (2013)**

# Binary Search Recap

```java
public static int binarySearch(char[] arr, char key) {
    int upper = arr.length-1;
    int lower = 0;
    while(upper >= lower) {
        int middle = (upper + lower)/2;
        if (arr[middle] == key) {
            return middle;
        }
        else if (arr[middle] > key) {
            upper = middle - 1;
        }
        else {
            lower = middle + 1;
        }
    }
    return -1;
}
```

- The binary search method applied to an array of characters
- SAMPLE OUTPUT:

```
Found at index: 0
Found at index: 6
Found at index: 2
Not found!
```

In the main method we call this function… how many times???

```java
char[] arr2 = {'a', 'c', 'g', 'i', 'm', 'x', 'z'};
int binarySearchResult = binarySearch(arr2, 'a');
System.out.println(binarySearchResult == -1 ? "Not found!" : "Found at index: " + binarySearchResult);
binarySearchResult = binarySearch(arr2, 'z');
System.out.println(binarySearchResult == -1 ? "Not found!" : "Found at index: " + binarySearchResult);
binarySearchResult = binarySearch(arr2, 'g');
System.out.println(binarySearchResult == -1 ? "Not found!" : "Found at index: " + binarySearchResult);
binarySearchResult = binarySearch(arr2, 'y');
System.out.println(binarySearchResult == -1 ? "Not found!" : "Found at index: " + binarySearchResult);
```

# Variable Length arguments list and the Arrays class - Recap

```java
public class Ex2and3 {
    public static void main(String[] args) {
        sayHello("Helen");
        sayHello("Helen2", "John1");
        sayHello("Helen3", "John2", "Diana", "George");
        System.out.println(Arrays.toString(new double[] {2.3, -1, -3, -4, 7}));
        System.out.println(Arrays.toString(negVals(2.3, -1, -3, -4, 7)));
    }//end main
    public static void sayHello(String... names ) {
        for(String name: names ) {
            System.out.print("Hello " + name + "!\t");
        }
        System.out.println();
    }//end sayHello
    public static double[] negVals(double...ds ) {
        int negCount = 0;
        for(int i = 0; i < ds.length; i ++) {
            if(ds[i] < 0) {
                ++negCount;
            }
        }
        double[] negatives = new double[negCount];
        for(int i = 0, j = 0; i < ds.length; i++) {
            if(ds[i] < 0) {
                negatives[j] = ds[i];
                j++;
            }
        }
        return negatives;
    }//end negVals
}//end class
```

- Sample Output:

```
Hello Helen!
Hello Helen2!   Hello John1!
Hello Helen3!   Hello John2!    Hello Diana!    Hello George!
[2.3, -1.0, -3.0, -4.0, 7.0]
[-1.0, -3.0, -4.0]
```

# Today's Agenda

- Introduction to Object Oriented Programming Paradigm.
  - Why OOP
  - Classes
  - Objects
- Introduction to ArrayList data structure.

# Let’s have a look at what we have done so far

- Small programs, most of them structured in one method – main method.
- All managed by a single individual programmer.
- They perform a small pre-specified range of tasks, such as take input, process the input and then output the results

```
import java.util.Scanner;
public class SmallProgram {
    public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            System.out.println("enter 2 numbers ");
            int a = sc.nextInt();
            int b = sc.nextInt();
            System.out.println("their sum is: " + (a+b));
            sc.close();
    } //end main
} //end class
```

- Such small programs are often best designed using… ***structured programming***

# Let's have a look at what we have done so far

- We have also learnt to decompose tasks into methods; for example

```
import java.util.Scanner;
public class SmallProgram2 {
    public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            System.out.println("enter 2 numbers ");
            int a = sc.nextInt();
            int b = sc.nextInt();
            printSumOf2Ints(a, b);
            sc.close();
            printSumOf2Ints(7, 14);
    } //end main
    public static void printSumOf2Ints(int a, int b){System.out.println("their sum is: " + (a+b)); }
} //end class
```

- Great practice, part of the modularisation process.
- But it doesn't go far enough: all the code still goes into a single class, and it is very hard to understand and maintain a program with hundreds of methods ...
- Also it does not allow team work.

# What about larger and more complex programs?

- Most computer programs are large and usually entail the collaboration of an entire team of programmers.
- They perform complex tasks and typically:
  - run continuously
  - respond to input from users and other programs
  - deliver output to users and other programs
- How can we then accomplish these???

# Object Oriented Programming

- we use OOP which overcomes problems associated with structured programming.
- **Object-oriented programming** = programming style in which tasks are solved by collaborating objects.
- Each object has its own set of data and methods that act upon that data.
- Already experienced this style when we used:

  String, Scanner, ArrayList etc. – each with its own set of data and methods

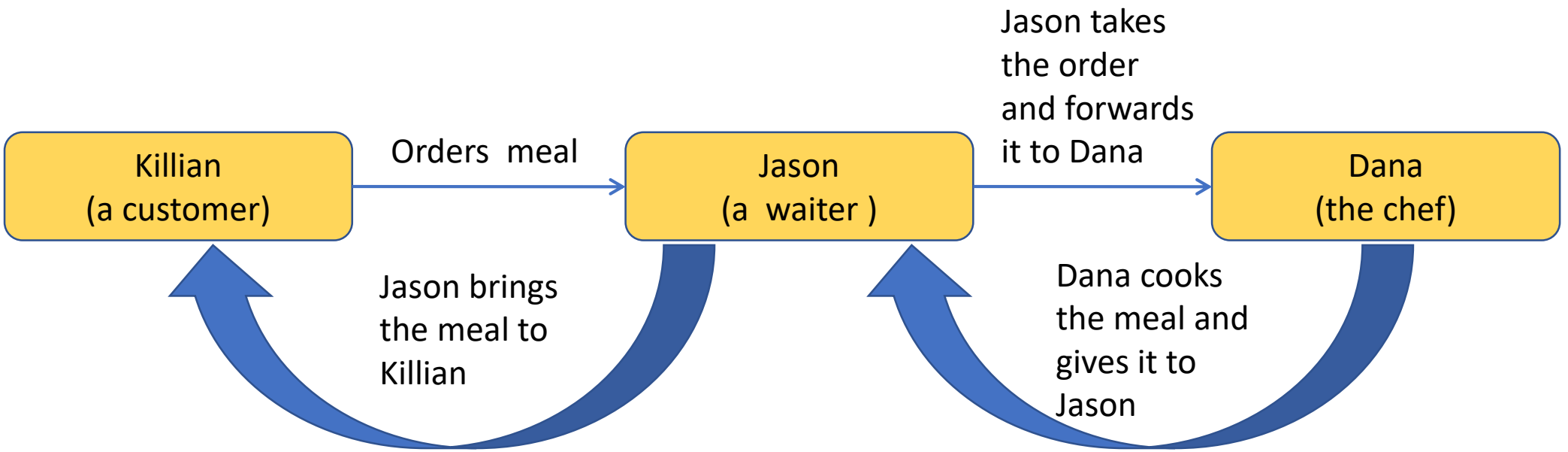
- Analogy to real world organisation example :

how do we organise a restaurant, for instance (we assume a very simple restaurant, but the same applies to complex ones)?

- Customers
- Waiters
- Chefs

- Each of these will interact repeatedly to obtain and provide services, respectively.

# Object Oriented Programming




And the entire process of interaction (the customer interacts with the waiter, the waiter interacts with the chef) repeats for other customers, other waiters and other chefs…

# Object Oriented Programming

- **OOP refers to the process of designing and implementing a co-operative community of interacting *objects*.**

  *in a RESTAURANT we encounter CUSTOMERS, WAITERS, CHEFS*

- each object provides a small number of relatively simple *services/behaviours*

  *customer ORDERS; waiter TAKES ORDERS; chef COOKS*

- objects *communicate* with each other to exchange information

  *customer interacts with the waiter, the waiter interacts with the chef and, conversely, the chef interacts with the waiter, the waiter interacts with the customer*

# Advantages of Object Oriented Programming

1. Complexity management
   - software development can be more reliably divided between independent groups → development can be broken down into manageable tasks;
   - Each task can then be managed by individual classes;
   - Objects/Instances interact with each other through pre-specified public ***interfaces***;
   - These interfaces are designed using best practices, such as the ***encapsulation*** and ***information hiding*** principles.
2. Reusable software
   - ***class libraries*** provide easy access to many standard services
   - developing ***software components*** that match the reliability and inter-changeability of hardware components
3. Natural modelling
   - problem identification, program design and program implementation all follow same process.

# Java and Object Oriented Programming

When you develop an object-oriented program, you create your own **objects** that describe what is important in your application.

**Objects** typically have:

- **Properties** implemented in java code as data fields/attributes
  for example, customers have names and favourite meals
- **Behaviours** implemented in java code as methods
  for example, customers order meals

# Why use classes?

- However, in Java, a programmer doesn't implement a single object.
- Instead, the programmer provides a **class** which describes a set of objects with the same properties and behaviour(s).

•Why???

•Let's look at the restaurant example:

➢there can be hundreds and hundreds of customers … should we implement code for each one of them?

➢NO – it would a very tedious, time consuming process;

➢Instead, we define a class that encapsulates the properties and behaviours of customers (such as name and ordering meals) and then create customer objects as needed using that class

# Why use classes …another example

- Assume you want to write code for representing 2 students: name David, year 1, student number B0001000, and name Aurelia, year 3, student number B0001200 :

```
//we can use  variables for each student
String nameOfStudent1 = “David”;
int yearOfStudent1 = 1;
String student1Number = “B0001000”;
String nameOfStudent2 = “Aurelia”;
int yearOfStudent2 = 3;
String student2Number = “B0001200”;
// what about 140 more students???
//– 420 more lines of code ☹
```

# Java and Object Oriented Programming

**Better to define these characteristics once, in a template and use that template every time we need to create a student.**

**Class** – defines a single concept applicable to a particular set of objects with specific properties/data specified as *variables* and behaviours specified as *methods*;

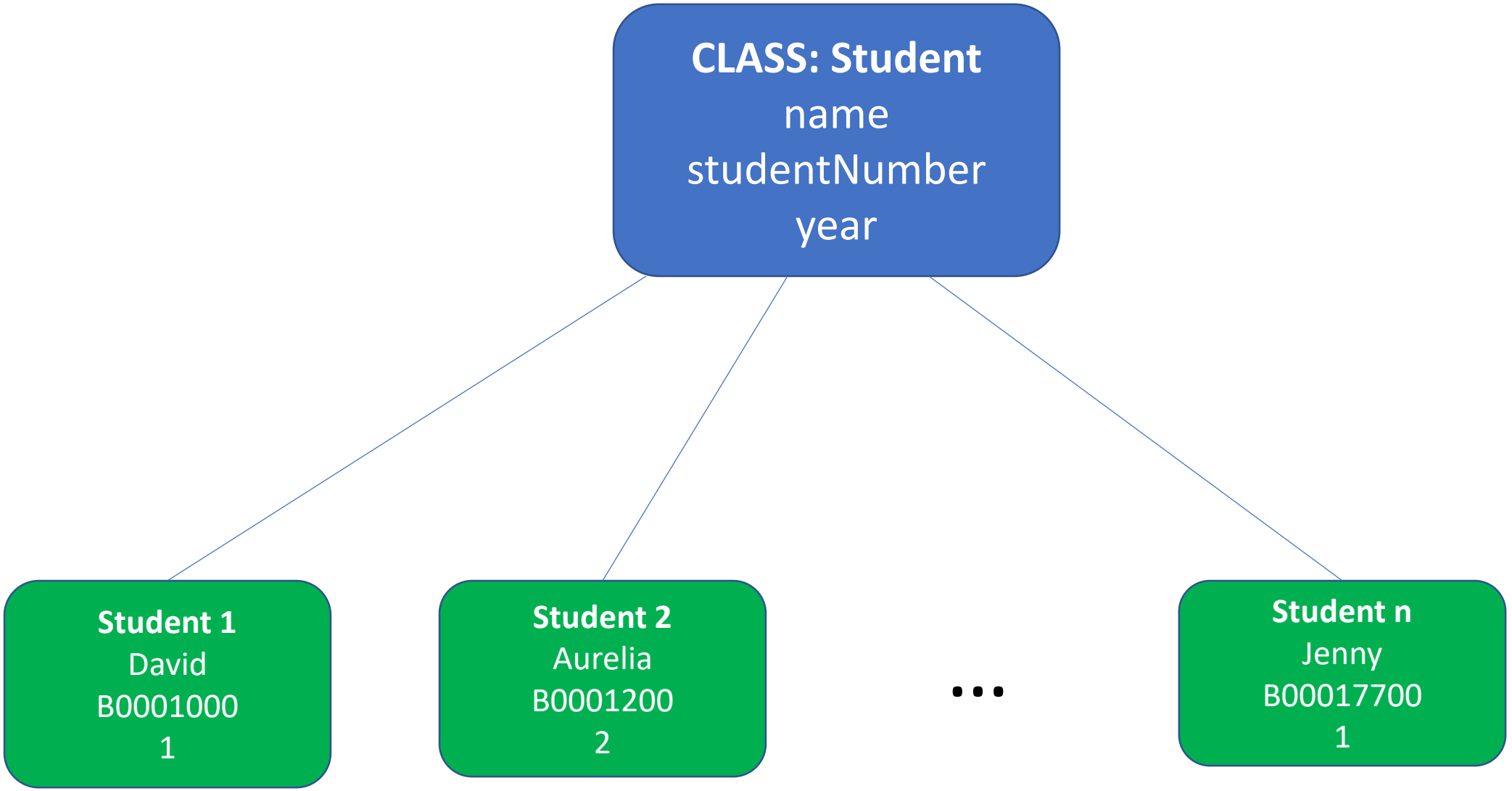
- It acts as **a blue print/template used to create objects**.

- We implement it using a class definition

**Object** – **is a single instance of a class** that is created by applying the given template with concrete data.

- You can create as many objects as needed by your application, using the same class/template

- in code, we create objects using class instantiation


CLASS: Student
name
studentNumber
year
Student 1
David
B0001000
1
Student 2
Aurelia
B0001200
2
...
Student n
Jenny
B00017700
1

# Class Definition

```
public class ClassName{
        (access modifier)   TYPE   variableName;
        …
}
```

- you can create a class by simply specifying the **class** keyword, followed by a **name**, followed by a pair of **{}**
- What goes inside the class is optional: you can have an empty class that compiles fine… but is really that useful???

```
public class Student{

        private String name;

        private String studentNumber;

        private int year;
}
```

Student class has 3 instance variables name, studentNumber and year

- Each Student object has a separate copy of these instance variables
- If you don't initialise these values to certain values, they will be initialised to default values….
- So name and Student number will be initialised to null, while year will be initialised to 0.
- NOTE: these variables should be made private, so other classes cannot modify them directly, that is they are well encapsulated

# Class Definition - what can we do with class so far?

```
public class Student{
        private String name;
        private String studentNumber;
        private int year;
} //does it compile??  --- yes
```

- Can we run it?
- No… it needs a main method to run it
- In fact, it is best practice to include a main method to test that your class functionalities work as they are supposed to work → typically the main method is used for unit testing/debugging purposes.

# Class instantiation?

```
// let's add the main method
public class Student{
        private String name;
        private String studentNumber;
        private int year;
    public static void main(String [] args){
        //instantiate/create 2 student objects
        Student student1 = new Student();
        System.out.println(student1.name);
        System.out.println(student1.studentNumber);
        Student student2 = new Student();
         System.out.println(student2.name);
        System.out.println(student2.year);
    }
} // what does it output??
```

- To instantiate or create an object we use the new keyword, to tell the compiler to allocate memory for it.
- To access an instance variable you need to use a given object variable name and the . (dot) operator:

student1.name

student1.year

```
null
null
null
0
```

So, how can we give them certain values????

# We can initialise the instance variables to values

```
// let's add the main method
public class Student{
        private String name = "David";
        private String studentNumber = "B0001000";
        private int year =  1;
    public static void main(String [] args){
        //instantiate/create 2 student objects
        Student student1 = new Student();
        System.out.println(student1.name);
        System.out.println(student1.studentNumber);
        Student student2 = new Student();
         System.out.println(student2.name);
        System.out.println(student2.year);
    }
} // what does it output??
```

```
David
B0001000
David
1
```

- What happens??

-All the variables will have the same values for all the objects that we create

-So using instant initialisation does not get us very far… so we will not use it

# We can use a constructor instead….

-Instead we will use a **constructor** which creates a new object and intitialises the instance variables for that object;

-It is similar to a method, but it **does NOT have a return type**;

-if you specify a return type the compiler will consider it a method;

-The constructor name must have the **same name as the class**

-**Syntax of a constructor in Java:**

```
public ClassName(parameter list){
        //code to initialise instance variables
}
```

-The access modifier is optional;

-The parameter list can be empty (and in this case is called **no arguments constructor**);

# We can use a constructor….

- The **no arguments constructor** initialises the instance variables to the same values for all the objects: default values chosen by the programmers:

```
public Student(){
        name  = “a name”;
        studentNumber = “a student number”;
        year = 1;
}
//in the main method
Student student1 = new Student();
System.out.println(student1.name + “ ” + student1.studentNumber + “ ”
                        + student1.year);
```

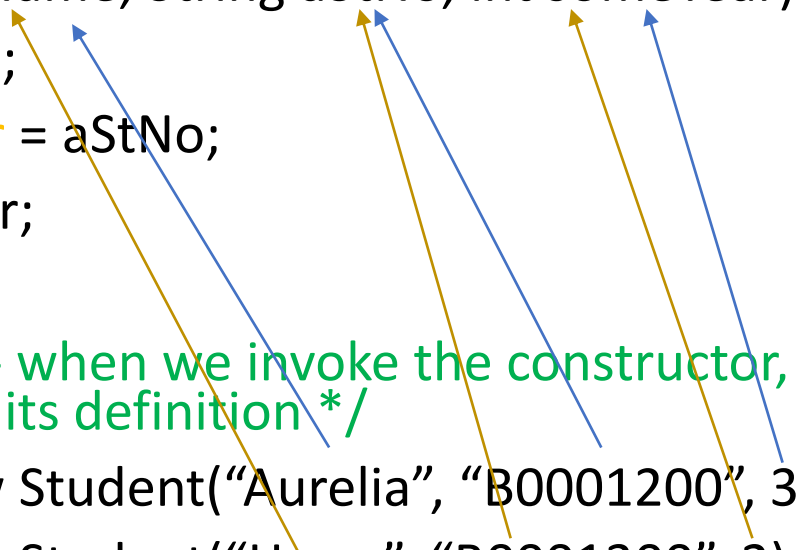
**Outputs:** a name a student number 1;

- and it would output the same for any other object created using that constructor..

# We can use a constructor that takes parameters instead….

- Like methods, you can have multiple overloaded constructors.
- The compiler selects the right one based on the argument list.

```
//let's add another constructor – note it also has the name of the class
public Student(String aName, String aStNo, int someYear){
        name  = aName;
        studentNumber = aStNo;
        year = someYear;
}
/* in the main method - when we invoke the constructor, the arguments passed will initialise the parameters specified in its definition */
Student student2 = new Student("Aurelia", "B0001200", 3);
Student student3 = new Student("Harry", "B0001300", 2);
System.out.println(student2.name + " " + student2.studentNumber + " "
                        + student2.year);
System.out.println(student3.name + " " + student3.studentNumber + " "
                        + student3.year);
```



**OUTPUT:**

Aurelia B0001200 3

Harry B0001300 2

```
 */
public class Student {
    private String name;
    private String studentNumber;
    private int year;

    // no arguments constructor
    public Student(){
        name = "a name";
        studentNumber = "a student number";
        year = 1;
    }
    //constructor with 3 parameters to initialise the 3 instance variables
    public Student(String aName, String aStNo, int someYear){
        name  = aName;
        studentNumber = aStNo;
        year = someYear;
    }
```

```
    /* main method to test and debug*/
    public static void main(String[] args) {
        //create a student object by using the no-arguments constructor
        Student student1 = new Student();
        //print the details to see if they have changed
        System.out.println(student1.name + " " + student1.studentNumber +
                " " + student1.year);
        //create 2 other students using the other constructor
        Student student2 = new Student("Aurelia", "B0001200", 3);
        Student student3 = new Student("Harry", "B0001300", 2);
        //print their details
        System.out.println(student2.name + " " + student2.studentNumber +
                " " + student2.year);
        System.out.println(student3.name + " " + student3.studentNumber +
                " " + student3.year);
    }//end main
}//end class
```

# More on constructors….

-The constructor  is called/invoked automatically when we use the new operator;

-How about about when we used the first time the new operator??? We did not have a constructor defined then… (see slide 17) and it still worked …

**If (and only if) you do not define a constructor, the compiler will define one automatically– the default constructor**

NOTE: a constructor typically initialises all instance variables.

- What about next year when we need to change the year only, or if Harry wants to be no longer associated with Harry Potter and wants to change his name to George??
- What if you want to change the values of an instance variable one at a time ??? Do we have to use the constructor to create a new object and change that one single value??
- NO… creating object adds overhead … so we should use the set methods !!!

# Use setter methods/mutators

```
/* we can add some methods to the Student class that allows us to change
instance variables one at a time*/
public void setName(String aName) {name = aName;}
public void setStudentNumber(String aStNum) {studentNumber = aStNum;}
public void setYear(int someYear){ year = someYear;}
//in the main method…
student3.setName(“George”);
student1.setStudentNumber(“B0001000”);
student2.setYear(4);
System.out.println(student3.name + “ ” + student1.studentNumber + “ ”
                                + student2.year); //what does it output???
```

George B000100 4

# Using setter methods/mutators

-Now we can make our variables take certain values for different objects.

-We can use these methods to modify/change/set new values for the data fields.

-These methods typically take as argument a variable that has the same type as the value of the field we are trying to modify

-These methods **are no longer static…they** are instance methods which means that we **invoke them using a concrete instance of the class**, no longer the name of the class, and it will change values for one single object at a time, not for all the objects

```
Student.setName("Joshua");
```

/* will not compile…because we need to specify a concrete object/instance for which we try to modify the name */

```
student3.setName("Joshua");
```

/* will compile and take the name instance variable is now changed to **Joshua** for student3 ONLY */

# Encapsulation and Information Hiding … exposing the public interface only

- What about accessing the properties of an object instantiated in a different class?? Can we access them directly??
- NO!!! – because we made private, so other user classes cannot modify them directly.
- So every class should have a **public interface:** a collection of methods through which the objects of a class can be manipulated.
- **ENCAPSULATION -** describes the mechanism of wrapping data/variables and methods acting upon that data into a single unit – the class, providing a public interface, while hiding the implementation details.
- Encapsulation enables **scalability**: changes in the implementation without affecting users of the class (by this we mean other classes that use/instantiate objects of that class);
- Encapsulation also enables **separation of concerns**: one class is concerned with one concept, another with a different concept, and so on… so if we modify one, it should not affect the rest.

# How can we use our class in other classes, though?

-Recall the way we use Scanner class to create instances in different classes

-We can do the same with our Student class:

- First we create a class called StudentGroup;
- In its main method we instantiate various Student objects;

```
public class StudentGroup{
        public static void main(String [] args){
                Student s1 = new Student("Joe Smith", "B000234567", 3);
                System.out.println(s1.name + " " + s1.studentNumber + " "
                                + s1.year);
        }
} //What can it output??
```

It actually does not compile, because the data fields name, studentNumber, year are private, and they cannot be accessed from a different class

# How can we use our class in other classes, though?

-We need a mechanism that will allow other classes to access the values of these fields safely, without directly being able to modify them.

-For this, we can use the **get methods/getters/accessors**

```
// we can add some methods to the Student class that allows us to do so
public String getName() { return name;}
public String getStudentNumber() {return studentNumber;}
public int getYear(){return year;}
```

-can be used in Student class as well as in other classes because they are public (note that the same applies to the set methods);

-They return the value of the instance variable specified in the name of the get method;

-Their parameter list is typically empty and they are also not static;

-So… the setters and the getters are part of the public interface of a given class.

# Let’s see what we have added to the Student class

```
// the setters/mutators
public void setName(String aName) {name = aName;}
public void setStudentNumber(String aStNum) {studentNumber = aStNum;}
public void setYear(int someYear){year = someYear;}

// the getters/accessors
public String getName() { return name;}
public String getStudentNumber() {return studentNumber;}
public int getYear(){return year;}
```

# Let's use/invoke the getters in a different class: the StudentGroup class…

```
public class StudentGroup{
        public static void main(String [] args){
                Student s1 = new Student("Joe Smith", "B000234567", 3);
                System.out.println(s1.getName()+ " " + s1.getStudentNumber() + " "
                                + s1.getYear());
                Student s2 = new Student("Jane Smith", "B00077777", 1);
                System.out.println(s2.getName()+ " " + s2.getStudentNumber() + " "
                                + s2.getYear());
                s2.setName("Jane O'Brien"); System.out.println(s2.getName());
        }
} //What can it output??
```

```
Joe Smith B000234567 3
Jane Smith B00077777 1
Jane O'Brien
```

# Let’s see what we have learnt so far

A class is like a template/blue print that we use to create concrete instances/ objects.

True

When we want to create an object/instance of a class we use the new operator…

True

A well encapsulated class does not declare its methods as public…

False

A constructor’s purpose is to initialise instance variables of a class…

True

Setter methods return the value of the  variable that appears in its name…

False

Getter methods are used to access and change the value of an instance variable…

False

# Let's see what we have learnt so far

- What is wrong with the following class?

```
public class Circle{
        private double radius;
        public Circle(double  aRadius){ aRadius = radius;}
        public int getRadius (){ return radius;}
        public void setRadius(double aRad){ radius = aRad;}
}
```

- The constructor does not initialise the variable radius to the value of parameter; it compiles, but the value of radius will still be… 0.0 (the default value)
- The return type of getRadius does not match the type we declared for radius; it does not compile, to fix it make the return type also double;

# What about output??

It is a bit tedious to repeatedly use print statements with the getters…Can you think of a way to address this issue???

**1. We can create  method printStudentDetails** and invoke it every time  we want to print the details associated with a particular Student instance (note, that this method is suitable only to console output);

```
// to the Student class add the following method
public void printStudentDetails (){
        System.out.println(“Student Details: ” + name + “, " + studentNumber +
                “, ” + year);
}
//in the main method we replace the print statements with calls of printStudentDetails
Student student2 = new Student("Aurelia", "B0001200", 3);
Student student3 = new Student("Harry", "B0001300", 2);
student2.printStudentDetails();
student3.printStudentDetails();
//but what will it print???
```

**It will print:**

Student Details: Aurelia, B0001200, 3

Student Details: Harry, B0001300, 2

# toString and other behaviours/methods

**2. The overridden method toString** - to use it with other methods of outputting not only console output

```
// to the Student class add the following method that returns the String representation of a given Student object
@Override
public String toString(){
        return “Student Details: ” + name + “, " + studentNumber + “, ” + year;
}
/* we don’t even need to call toString explicitly to output to console; in the main method we can pass the student objects to the println method and the compiler will automatically invoke on our behalf */
```

**System.out.println(student2); System.out.println(student3);**

- We can add many other methods… as many methods as we need.

Think of some more behaviours that characterise students only …

- Studying (hmmm…???)
- Partying (this applies to most of you, though… ☺)
- Going to the lectures (hmm…again a rather infrequent behaviour of students)
- And many others… but we should add methods for them only if we need

# More on Instance vs. Static…

- **Instance variables** are initialised when an object is created; **static variables** belong to the whole class, and do not depend on any instance/object.
- **Static variables** are loaded in memory when the class is loaded; their values are the same for all instances/objects of that class.
- **Instance variables** cannot be accessed by static methods, because, if you recall, static methods definitions are loaded into memory when the class is loaded, but instance variables are loaded only when we create a given object → **Static methods** can only access static variables.
- **Instance methods** can access both instance variables and static variables.
- **Static variables and methods** can be invoked using either the class name or a particular instance of that class; however is recommended to use class name.

# Instance vs. static

-Let's add to our class a static variable and modify its constructors accordingly

```
//add the following variable to the Student class
static int numberOfStudentsCreated;
//modify the constructors --- how??
public Student(){
   name = "a name";
   studentNumber = "a student number";
   year = 1;
    numberOfStudentsCreated ++;
}
public Student(String aName, String aStNo, int someYear){
   name  = aName;
   studentNumber = aStNo;
   year = someYear;
    numberOfStudentsCreated ++;
} //so what happens when we invoke the constructors???
```

**Every time we invoke either constructor, the static variable will be increased by 1**

# Instance vs. static

-Let's see how is this outputted in the main method

```
//in the main method of the Student class
Student student1 = new Student();
System.out.println(Student. numberOfStudentsCreated);
//the above statement outputs 1
Student student2 = new Student("Aurelia", "B0001200", 3);
System.out.println(student2.numberOfStudentsCreated);
//the above statement outputs 2
Student student3 = new Student("Harry", "B0001300", 2);
System.out.println(student1.numberOfStudentsCreated);
//the above statement outputs 3
```

So the numberOfStudentsCreated will be increased every time we use new keyword, and that increase is reflected in all the instances of the Student class

```java
public class Student {
    private String name;
    private String studentNumber;
    private int year;
    static int numberOfStudentsCreated;

    // no arguments constructor
    public Student(){
        name = "a name";
        studentNumber = "a student number";
        year = 1;
        numberOfStudentsCreated++;
    }
    //constructor with 3 parameters to initialise the 3 instance variables
    public Student(String aName, String aStNo, int someYear){
        name  = aName;
        studentNumber = aStNo;
        year = someYear;
        numberOfStudentsCreated++;
    }
```

```java
// the setters/mutators
public void setName(String aName) {name = aName;}
public void setStudentNumber(String aStNum) {studentNumber = aStNum;}
public void setYear(int someYear){year = someYear;}

// the getters/accessors
public String getName() { return name;}
public String getStudentNumber() {return studentNumber;}
public int getYear(){return year;}

//method to print details of a given student
public void printStudentDetails(){
    System.out.println("Student Details: " + name + ", " + studentNumber +
            ", " + year);
}
```

```
    /* main method to test and debug*/
    public static void main(String[] args) {
        //create a student object by using the no-arguments constructor
        Student student1 = new Student();
        //pritn how many objects have been created so far
        System.out.println(Student.numberOfStudentsCreated);
        //print the details to see if they have changed
        student1.printStudentDetails();
        //create 2 other students using the other constructor
        Student student2 = new Student("Aurelia", "B0001200", 3);
        System.out.println(student2.numberOfStudentsCreated);
        Student student3 = new Student("Harry", "B0001300", 2);
        System.out.println(student1.numberOfStudentsCreated);
        //print their details
        student2.printStudentDetails();
        student3.printStudentDetails();
    }//end main
}//end class
```

# OUTPUT

```
1
Student Details: a name, a student number, 1
2
3
Student Details: Aurelia, B0001200, 3
Student Details: Harry, B0001300, 2
```

# More questions…

```
public class Person{
        private String name;
        public Person(String  aName){ name = aName;}
        //…
        public static void main(String args[]){
                Person p1 = new Person("John");
                Person p2 = new Person();
        }
}
```

-What is the value of the variable *name* for p1?

John

-What is the value of the variable *name* for p2?

The code will not compile for p2 because we have not defined a constructor that takes no arguments.

-Assume we try to fix the code as follows: **Person p2 = new Person(p1.name);** does it compile, and ,if so, what will be the value of the variable *name* for p2 now??

John

# ArrayLists

- One big **disadvantage of the array data structure** is that you cannot change the size of an array once created → you cannot add or remove elements from an array
- In situations when you don't know how many elements the data structure should hold, arrays are not suitable.
- Better suited is the **ArrayList** generic class, found in **java.util** package;
- It has **2 important advantages**:

1. You can add or remove elements as needed, as array lists can dynamically grow or shrink
2. The ArrayList class defined in java api offers many methods that allows us to process and manipulate arrays.

# Array Lists - syntax overview for declaring, creating and using common methods

*Syntax*

To construct an array list: `new ArrayList<typeName>()`

To access an element: *arraylistReference*`.get(index)`
*arraylistReference*`.set(index, value)`

Variable type — Variable name — An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

The add method appends an element to the array list, increasing its size.

Use the get and set methods to access an element.

The index must be ≥ 0 and < friends.size().

# ArrayList class

- One of the most used collection classes;
- It combines array features with those of a list;
- It's elements are also indexed from 0 to size of the list-1:

If size of the list is **10**: first index is **0**, last index is **9**

- **Common operations** of a list:
  - **Add item(s) to a list;**
  - **Modify an item;**
  - **Delete items from a list**
  - **Retrieve an item from a list**
  - **Iterate over the items**

# How to create an ArrayList

- unlike arrays, you don't need to specify the size, although you could specify an initial capacity by using the corresponding overloaded constructor;

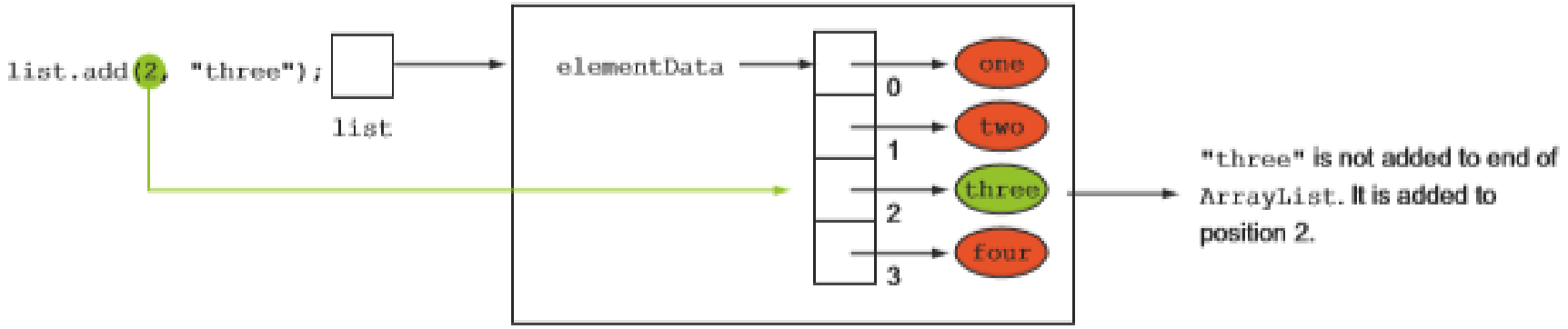
```
ArrayList<String> stringList = new ArrayList<>();/*constructs an empty list that can hold only String objects, with an initial default capacity of 10*/

ArrayList<Student> studentList = new ArrayList<Student>();/*constructs an empty list that can hold only Student objects, also with an initial default capacity of 10; can also specify the type on the right side, but not required*/

ArrayList<Integer> intList = new ArrayList<>(7);/*constructs an empty list that can hold only Integer objects, with an initial capacity of 7 */
```

# Adding elements to an ArrayList

We can use the overloaded versions of the add method:

1. One that adds an object to the end of the list;
2. One that adds an object at the specified index.

```
ArrayList<String> list = new ArrayList<>();
list.add("one"); //adds one
list.add("two"); //adds two after one at the end
list.add("four"); // adds four after two at the end
list.add(2, "three"); /* adds three after two, at index 2, and four is shifted to the right one place */
```



# Accessing elements of an ArrayList

- We can access the elements of a list (one at a time) by using the index with the method get:

```
/* using the previous list  and an enhanced for loop */
for(int j = 0; j < list.size(); j++){//note the method size(), not length()
        System.out.println(list.get(j));
}
 //prints the elements: one, two, three, four on different lines
```

- We can use an enhanced for loop also:

```
for(String element : list){
        System.out.println(element);
}
//prints the elements: one, two, three, four on different lines
```

- We can also print an entire list:

```
System.out.println(list);
//prints [one, two, three, four]
```

# Other useful methods from ArrayList

- **size method –** returns the number of elements in the list

```
/* using the previous list: [one, two, three, four] */
System.out.println(list.size());  // prints 4
```

- **contains method** – returns true or false, depending on whether the list contains the specified element

```
/* using the previous list*/
System.out.println(list.contains("one")); //prints true
System.out.println(list.contains("five"));
//prints false
```

- **We can modify an existing element using the set method** – replaces the value of the element of the specified index with the specified value;

```
/* using the previous list */
list.add("one"); // so the list is now [one, two, three, four, one]
list.set(4, "five"); //replaces "one" at index 4 with "five"
System.out.println(list.get(4));
//prints "five" because the list is now [one, two, three, four, five]
```

# Deleting elements of an ArrayList

- **We can use the remove method** – which has 2 overloaded versions:

1. remove(int index) – removes the element at the specified index/position

```
/* using the previous list which now is: [one, two, three, four, five] */
list.remove(1); //removes “two” from the list
System.out.println(list);
//prints [one, three, four, five]
```

2. remove(Object obj) – removes the first occurrence of the specified element;

```
list.add(“three”);
// so the list is now: [one, three, four, five, three]
list.remove(“three”)); //removes the first instance of “three”, at index 1
System.out.println(list);
//prints [one, four, five, three]
```

# ArrayList – limitations/disadvantages

1. Does NOT allow primitive data types:

```
ArrayList<int> integerList = new ArrayList<>(); // won't compile
```

- But it gets around this issue by using the wrapper classes: if you want the list to hold numeric types, they need to be of **a wrapper type**:

  **I**nteger, **D**ouble, **B**oolean, etc.

- note the beginning capitals:

```
ArrayList<Integer> integerList = new ArrayList<>();
ArrayList<Double> doubleList = new ArrayList<>();
ArrayList<Boolean> booleanList = new ArrayList<>();
```

2. Although is fast to search, it is slow to insert and delete…

# Wrapper Classes

| Primitive Type | Wrapper Class |
|---|---|
| byte | Byte |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

- The Wrapper classes allow **automatic conversion of primitive data types** to their corresponding wrapper type - autoboxing:

```
Double doubleWrapped = 2.5;
Integer intWrapped = 7;
Boolean booleanWrapped = false;
```

- And vice versa: automatic conversion of wrapper types to primitive types- auto-unboxing :

```
ArrayList<Double> dVals = new ArrayList<>();
dVals.add(3.4); /*auto-boxing: will first convert the primitive double type to a Double type, and then adds it to the list */
double doubleVal = dVals.get(0); /*auto-unboxing: will first convert the wrapper Double type to a primitive double type and assigns it to the variable “doubleVal”*/
```

| | |
|---|---|
| `ArrayList<String> names = new ArrayList<String>();` | Constructs an empty array list that can hold strings. |
| `names.add("Ann");`<br>`names.add("Cindy");` | Adds elements to the end. |
| `System.out.println(names);` | Prints `[Ann, Cindy]`. |
| `names.add(1, "Bob");` | Inserts an element at index 1.<br>`names` is now `[Ann, Bob, Cindy]`. |
| `names.remove(0);` | Removes the element at index 0.<br>`names` is now `[Bob, Cindy]`. |
| `names.set(0, "Bill");` | Replaces an element with a different value.<br>`names` is now `[Bill, Cindy]`. |
| `String name = names.get(i);` | Gets an element. |
| `String last = names.get(names.size() - 1);` | Gets the last element. |
| `ArrayList<Integer> squares = new ArrayList<Integer>();`<br>`for (int i = 0; i < 10; i++)`<br>`{`<br>`  squares.add(i * i);`<br>`}` | Constructs an array list holding the first ten squares. |

```java
import java.util.ArrayList;

/**
 * @author Aurelia Power
 *
 */
public class ArrayListDemo {
    public static void main(String[] args) {
        //create an empty list
        ArrayList<String> friends = new ArrayList<>();
        System.out.println("Size of friends list before adding any people: "
                + friends.size()); //size is 0
        //add some friends
        friends.add("John");
        friends.add("Helen");
        friends.add("Garry");
        //print the contents of the list and its size
        System.out.println("Friends list (so far...): "
                + friends);
        System.out.println("Size of list after adding John, Helen, Garry: "
                + friends.size());//3
        //I got a new best friend, so I add it to the front
        friends.add(0, "Brendan NewBestFriend");
        System.out.println("After adding the new best friend, my list is: "
                + friends);
        System.out.println("Size of friends list is now: "
                + friends.size());//4
```

```
        //change the name from John to Joe
        friends.set(1, "Joe Smith");
        System.out.println("After correcting the name to Joe: "
                + friends);
        System.out.println("Size of friends list is still: "
                + friends.size());//4
        //Garry crossed me, so I remove him from my list of friends
        friends.remove("Garry");
        System.out.println("Garry upset me, so I "
                + "remove him from the list. The list has now: "
                + friends);
        System.out.println("Size of my friends list is now: "
                + friends.size());//3
        //remove a friend at the given index
        friends.remove(1);
        System.out.println("After a while Joe also crossed me, so I "
                + "remove him from the list. The list has now: "
                + friends);
        System.out.println("Size of frinds list is now: "
                + friends.size());//2
    }//end main method
}//end class
```

# ArrayList - Exercises

- **What is wrong (if anything) with this code snippet?**

```
ArrayList<String> names;
names.add(“Bob”);
```

We haven’t created the actual list; the correct way is:

```
ArrayList<String> names = new ArrayList<>(); // then add “Bob”
```

- **What does the array list names contain after the following statements?**

```
ArrayList<String> names = new ArrayList<>();
names.add("Bob");
names.add(0, "Ann");
names.remove(1);
names.add("Cal");
```

[Ann, Cal]