Fundamentals of Programming 2

Lecture 9

Aurelia Power, TU Dublin – Blanchardstown Campus, 2019

*** Notes based on the Java Oracle Tutorials (2019), Deitel & Deitel (2015), and Horstman (2013)**

Object Oriented Programming - RECAP

- We use OOP to overcome 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.
- 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/instance variables: for example, customers have names and favourite meals
- Behaviours implemented in java code as methods:

  for example, customers order meals

Advantages of OOP - RECAP

1. Complexity management
   - ***Using encapsulation*** and ***information hiding*** principles, objects should interact *only* through pre-specified ***interfaces***
   - software development can be more reliably divided between independent groups
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 OOP - RECAP

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

Typical definition and members of a class- RECAP

```
public class ClassName{
        // static  variables that apply to all objects instantiated using this class
        (access modifier)   static TYPE  variableName1;
        …
       // instance variables /data fields/attributes/properties – each object will have a different copy
       (access modifier)   TYPE   variableName2;
        …
       //(overloaded) constructor(s)
       (access modifier) className(parameterList){
                //code to typically initialise instance variables
       }
       …
       //behaviours/methods
       (access modifier)(static) RETURN_TYPE   methodName( parameterList){
                //code to implement the method
       }
       …
}
```

Typical definition and members of a class- RECAP

- class keyword and the name of the class (chosen by us to represent an entity) are required in order to define a class;

- **variable(s)** represent attributes/properties that apply to the given entity that we want to represent; they are optional.

- **method(s)** represent behaviours that apply to the given entity that we want to represent; are also optional.

- **constructor(s)** has the role of initialising the instance variables of an 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;** are also optional to define: if we don't define one, java will define the default one implicitly.

-Instance variables should be specified as private, while methods should be typically specified as public – the principles of information hiding and encapsulation.

-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.

ArrayList data structure - RECAP

- An array list stores a sequence of values whose size can change.
- An array list offers **two significant advantages**:
  - ✓ Array lists can grow and shrink as needed.
  - ✓ The ArrayList class supplies methods for common tasks, such as inserting and removing elements.

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");
```

Use the `get` and `set` methods to access an element.

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

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

Comparing array and array list operations

Assuming the following, we can compare the array structure vs arrayList structure along the main aspects:

- an array of integers called **values**
- an array list of integers called vals:

Operation	Arrays	Array Lists
Get/access an element	x = values[4];	y = vals.get(4);
Replace/Modify an element	values[5] = 37;	vals.set(5, 37);
Obtaining the number of elements	values.length	vals.size();
Remove an element	NA – array size is final	vals.remove(5)
Add an element at the end	NA – array size is final	vals.add(46)
Insert an element	NA – array size is final	vals.add(1, 77)
Working with numbers	Arrays can hold primitives or wrapper references	ArrayLists can only have wrapper references

Let's see how we can apply the ArrayList…

```java
/**
 * @author Aurelia Power
 *
 */
public class ShopApp {
    public static void main(String[] arguments) {
        //create an empty list
        ArrayList<String> products = new ArrayList<>();
        //add initial products
        products.add("eggs");
        products.add("oranges");
        products.add("smoked ham");
        products.add("bread");
        products.add("lettuce");
        //print the list
        System.out.println("Available products: " + products);
        //fix the error and re-display the list
        products.set(2, "cured ham");
        System.out.println("Available products: " + products);
        //run out of oranges: remove that product, and re-display the list
        products.remove(1);
        System.out.println("Available products: " + products);
        //add the best product at the top of the list
        products.add(0, "BROCCOLI");
        System.out.println("Available products: " + products);
    }//end main
}//end class
```

OUTPUT:

Available products: [eggs, oranges, smoked ham, bread, lettuce]

Available products: [eggs, oranges, cured ham, bread, lettuce]

Available products: [eggs, cured ham, bread, lettuce]

Available products: [BROCCOLLI, eggs, cured ham, bread, lettuce]

- And that's one of the exercises from lab8…

```java
/**
 * class that encapsulates a Car representation
 * */

/**
 * @author Aurelia Power
 *
 */
public class Car {
    // instance variables/attributes
    private String make, regNo;
    private double engSize;

    // 2 overloaded constructors: a no-args one to initialise attributes to default
    // values, and one that takes 3 parameters to initialise them to custom values
    public Car() {
        make = "Dacia"; // the default make is Dacia :)
        regNo = "191-D-1";// the default is the first car registered this year
        engSize = 0.1; // the default value is very small for environmental reasons
    }
    public Car(String m, String r, double e) {
        make = m; // could be any make, depends on the argument passed for parameter m
        regNo = r; // could be any registration, depends on the argument passed for parameter r
        engSize = e; // could be any size, depends on the argument passed for parameter e
    }
```

```java
// setters/mutators
public void setRegNo(String regNo) {
    this.regNo = regNo;
}
public void setMake(String make) {
    this.make = make;
}
public void setEngSize(double engSize) {
    this.engSize = engSize;
}

//getters
public String getRegNo() {
    return regNo;
}
public String getMake() {
    return make;
}
public double getEngSize() {
    return engSize;
}
```

```java
// instance method to print a Car object details; outputs to console only
public void printCarDetails() {
    System.out.println("Car Object <" + make + ", " + regNo + ", "+ engSize + ">");
}

// override the instance method toString to get a string representation of a Car object
// works with any output type
@Override
public String toString() {
    return "Car Object <" + make + ", " + regNo + ", "+ engSize + ">";
}
```

```java
// the main method to do unit testing
public static void main(String[] args) {
    // use the no-args constructor to create a Car object
    Car car1 = new Car();
    // print the details of the car using the method printCarDetails
    car1.printCarDetails();
    // modify the make
    car1.setMake("Some other cheap make...");
    // modify the engine size
    car1.setEngSize(0.99);
    // print the modifications using the get methods
    System.out.println("After mofifying make and engine size of car1: " + car1.getMake() +
            ", " + car1.getEngSize());
    // re-print the details of the car using the method printCarDetails
    System.out.print("After making modifications to car1: ");
    car1.printCarDetails();

    System.out.println("-----------------------------------------------------------------");

    // use the other constructor to create another Car object
    Car car2 = new Car ("Range Rover", "191-D-7", 5.7);
    // print the details of the Car using the method toString which in a print statement
    // does not even need to be explicitly called... the compiler will do so
    System.out.println(car2);
    // modify the regNo
    car2.setRegNo("191-MH-7");
    // print the modification using the get method
    System.out.println("After modifying reg for car2: " + car2.getRegNo());
    // re-print the details of the car using the toString method... let's invoke it
    // explicitly this time
    System.out.println("After making modifications to car2: " + car2.toString());
}
```

```
Car Object <Dacia, 191-D-1, 0.1>
After mofifying make and engine size of car1: Some other cheap make..., 0.99
After making modifications to car1: Car Object <Some other cheap make..., 191-D-1, 0.99>
--------------------------------------------------------------------
Car Object <Range Rover, 191-D-7, 5.7>
After modifying reg for car2: 191-MH-7
After making modifications to car2: Car Object <Range Rover, 191-MH-7, 5.7>
```

…and that’s another exercise from lab 8

Is there a limit to how many objects we can create?? Not really…

- We can create as many objects of type Car as we need using one of the 2 constructors that we have defined.
- To invoke a constructor, we need to use the new operator and then the name of the constructor, and pass as arguments values corresponding to each of the parameters. For example:

```
Car car1 = new Car();

car1.printCarDetails();

Car car2 = new Car("opel", "12 OY 123", 3.4);

System.out.println(car2);

Car car3 = new Car("nissan", "07 D 123" );

/*it will not compile: there is no constructor defined in the class Car that takes 2
   parameters.*/
```

What about creating Car objects outside the Car class??

```
import java.util.ArrayList;
/**
 * @author Aurelia Power
 *
 */
public class CarShow {
    public static void main(String[] args) {
        // create an empty list to hold Car object references
        ArrayList<Car> cars = new ArrayList<>();
        // create 3 cars and add them to the list
        Car c1 = new Car(); cars.add(c1);
        Car c2 = new Car("Mazda", "172-C-123", 3.4); cars.add(c2);
        cars.add(new Car("Nissan", "151-WW-345", 3.4));
        // display the list to the customers
        System.out.println("Initially: " + cars);
        // typed the wrong size engine for one of the cars, say last car, so we need to correct the mistake.
        cars.get(2).setEngSize(4.3);
        // Redisplay the list to the customers.
        System.out.println("After correction: " + cars);
        // One customer bought a car say the nissan, so you should remove it from the list.
        cars.remove(1);
        // Redisplay the list to the customers.
        System.out.println("After purchase: " + cars);
    }
```

Another exercise from lab 8…

```
Initially: [Car Object <Dacia, 191-D-1, 0.1>, Car Object <Mazda, 172-C-123, 3.4>, Car Object <Nissan, 151-WW-345, 3.4>]
After correction: [Car Object <Dacia, 191-D-1, 0.1>, Car Object <Mazda, 172-C-123, 3.4>, Car Object <Nissan, 151-WW-345, 4.3>]
After purchase: [Car Object <Dacia, 191-D-1, 0.1>, Car Object <Nissan, 151-WW-345, 4.3>]
```

Today's Agenda

- Introduction to Object Oriented Programming Paradigm continued:
  - More on the need for private variables
  - More on Object References
  - More on Strings

Public and private access modifier

- If you recall, we said that we should make our variables private to avoid other classes changing them directly … why?
- Let's assume the following small classes:

```
public class Person{
        public int age;
}
/*can access directly the
variable age from a different
class, for example, Family
where you can pass any value;
but in fact age should only have
positive values */
```

```
public class Family {
    public static void main(String[] args) {
        //the first person is mum
        Person mum = new Person();
        mum.age = 34;
        System.out.println(mum.age);
        //the second person is dad
        Person dad = new Person();
        dad.age = -37;//by mistake I used a negative value
        System.out.println(dad.age);
        //the third person is the baby
        Person baby = new Person();
        System.out.println(baby.age);
    }
}
```

Public and private access modifier

- We can avoid this by making our variable **private** and impose additional restrictions on how that variable is changed using the **public interface,** that is , the constructor and the set and get methods.
- Let's modify our class as follows:

```
public class Person {
    //instance variable
    private int age;
    //constructor
    public Person(int ageIn){
        if(ageIn >= 0){
            age = ageIn;
        }
    }
    //the setter
    public void setAge(int newAge){
        if(newAge >= 0){
            age = newAge;
        }
    }
    //the getter
    public int getAge(){
        return age;
    }
    //print person details to console
    public void printPersonDetails(){
        System.out.println("Person's age: " + age);
    }
}
```

What will happen to the Family class?

Will no longer compile, because it tries to access the private variable age

So we need to change it…

Exercise: Override the method toString in the class Person.

Let’s modify the Family class and its main method

```
//the first person is mum
Person mum = new Person(34);
System.out.println(mum.getAge());
//the above statement outputs 34
//the second person is dad
Person dad = new Person(-37); //by mistake I used a negative value
System.out.println(dad.getAge());
//the above statement outputs 0; not -37
//let’s change dad’s age
dad.setAge(37);
dad.printPersonDetails(); //now it outputs: Person’s age: 37
//the third person is the baby
Person baby = new Person(0);
System.out.println(baby.getAge());
//the above statement outputs 0
```

Object References

- We have already touched this topic when we talked about arrays references because arrays are not primitive data types…
- Recall that we said that when we invoke a constructor it allocates memory and creates a new object:

  Person per = new Person(25);

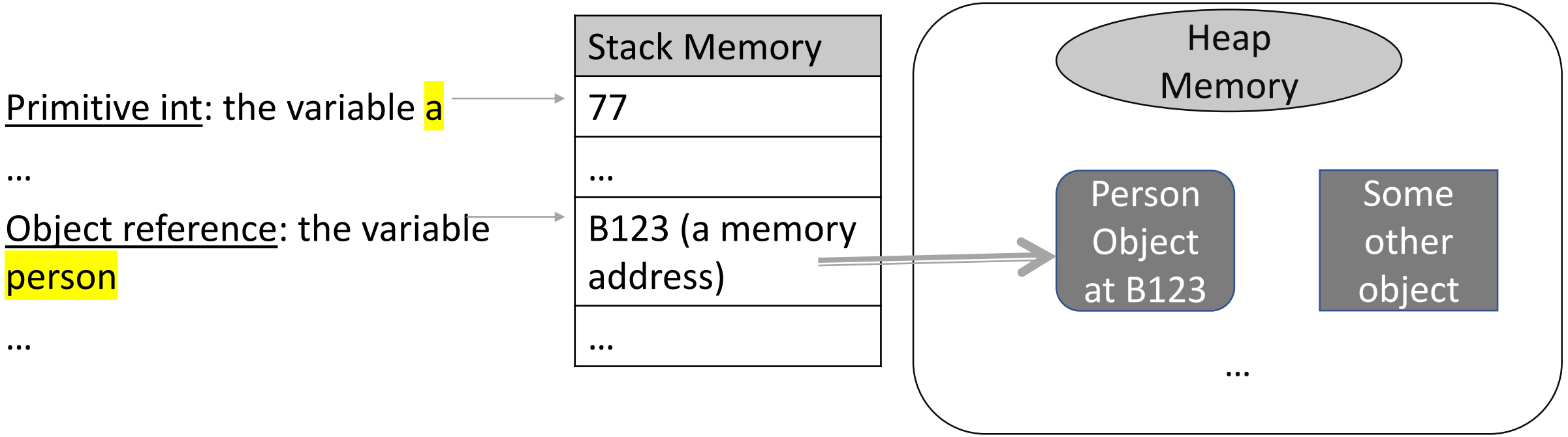
- So what does the variable per hold?
- It does not hold an object, but the memory location of the object also called **object reference**; this is stored on the **stack memory,** along with variables that hold primitive values.
- The object itself is stored elsewhere, on the **heap memory**;
- So, when a variable contains the memory location of an object we say that it **refers to an object.**

Object References

Let's consider the following code in the main method of a class:

int a = 77; Person person = new Person(age); //Does it compile???

Yes, but the way age and person object are represented in memory is different

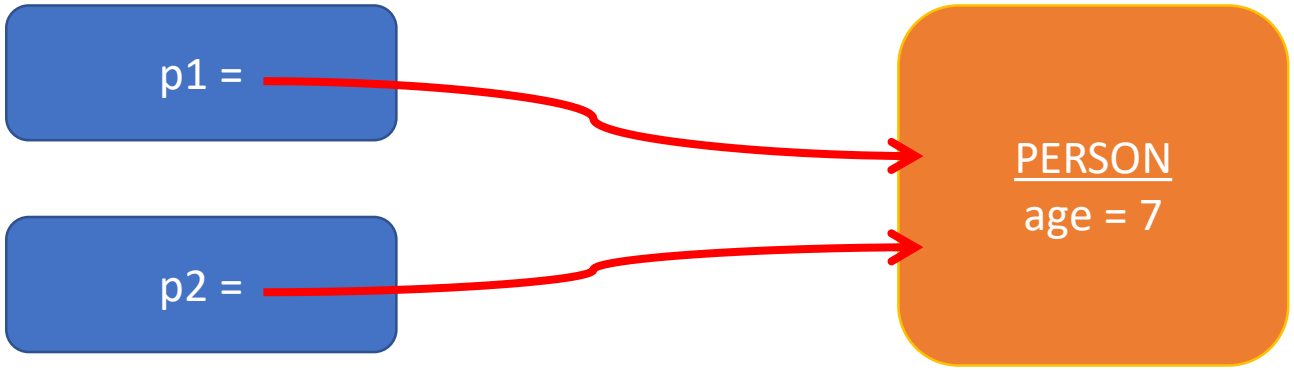


Object References

- The way Java represents variable references and objects in memory allows 2 or more variables to store references to the same object:

```
Person p1 = new Person(7);

Person p2 = p1;
```



- So p1 and p2 refer to the same object … so that means that if we change the value of age for p1, that will also be reflected in p2

```
p1.setAge(25);

p1.printPersonDetails();

//prints: Person's age: 25

p2.printPersonDetails();

// also prints Person's age: 25
```

Object References vs. Primitives – some important differences

- **Storage:**

➢Object reference variables store the address of the object (or references)

➢Primitive variable store the actual values;

- **Default values:**

➢Object reference variables have one literal value: null

➢Primitive variable have different literal values: int 0, decimal 0.0, char /u0000, etc.

- **Copying:**

➢When we copy an object reference, both the original and the copy refer to the same object

➢When we copy a primitive value the original value and the copy are independent.

Exercise time:

```
/* Assume class and main method
   declarations; also assume we
   use the previous Car class*/


Car c1 = new Car();
double size = 3.4;
double size2 = size;
size2 = 2.4;
Car  c2 = c1;
c2.setEngSize(size);
```

Does it compile?

yes

What is the final value of size2?

2.4

What is the final value of size?

3.4

What is the final value of the engine size of c2 (i.e., c2.getEngSize())?

3.4

What is the final value of the engine size of c1 (i.e., c1.getEngSize())?

3.4

Strings – again…

- A String is a **sequence of characters**;
- A string is NOT a primitive data type;
- A String value is denoted by double quotes:

“This is a String”

- Strings are created in Java using various techniques, the following **are common ways** to create them:
  - Using literal values:

    String s = “This string”;

  - Using the new operator:

    String s2 = new String(“This string”);

Strings - intro

- You can assign a null value to a String object reference:

String s3 = null;

- **NOTE:** an empty string is different from null

String str = “”; //is not null

- String objects are **immutable**: once created you cannot change them, because a string is backed by a final array of characters which you can obtain it using the **toCharArray()** method:

```
String s = “hello”;
char[] chars = s.toCharArray(); //the array contains: h, e, l, l, o
```

- Java provides great support for String handling and processing
- We will discuss some..
- For others, check the API…

Strings - equality

- If you want to find out whether two Strings have the same characters sequence you MUST use the **equals method** defined in the String class, which returns a boolean value;
- **== operator** will only compare the string objects' memory addresses (to see if they refer to the same object).
- **s1.equals(s2)** – returns true if the strings s1 and s2 contain the same sequence of characters; if not, it returns false

Examples:

"fop1".equals("fop2"); // returns false

"fop2".equals("fop2"); // returns true

- There is also a version that compares 2 strings irrespective of capitalization**:** **s1.equalsIgnoreCase(s2)**

Examples:

"fop1".equalsIgnoreCase("fOP2"); // returns false

"foP2".equals("FOp2"); // returns true

Strings equality - another example

```
String s1 = new String(“hello”);
 String s2 = new String(“hello”);

 if(s1.equals(s2)) {
        // the condition is true so the statement inside the if block will be executed
          System.out.println(“they have same value”);
 }
 if(s1 == s2) {
        // the condition is false so the statement inside the if block will NOT be executed
         System.out.println(“they refer to the same object”);
 }
```

Strings - comparisons

- If we want to compare 2 strings in terms of lexicographical/alphabetical order we can use the **compareTo** **method** defined in the String class, which **returns an int value**:
  - **0** if the first string and the second string have exactly the same sequence
  - A negative value if the first string comes before the second string in alphabetical order
  - A positive value if the first string comes after the second string in alphabetical order

NOTE: There is also a **compareToIgnoreCase** version which disregards case differences.

s1 = “str**i**ng”; s2 = “str**o**ng”;

System.**out.println(s1.compareTo(s2)); //prints a negative value**

Strings comparisons – an example

```
System.out.println(s2.compareTo(s1));
//prints a negative value
System.out.println(s2.compareTo("strong"));
// prints 0

if(s1.compareTo(s2) < 0) {

/*the boolean condition is true because i comes before o, so s1.compareTo(s2) will yield a
negative value and the next statement will execute*/

        System.out.println("s1 comes before s2" ); }

if(s2.compareTo(s1) > 0) {

/* the boolean condition is also true because o comes after i, so s2.compareTo(s1) will yield
a positive value and the next statement will execute */

        System.out.println("s2 comes after s1" ); }
```

Strings – number of characters

- You can find out how many characters are in a String using the **length()** method (NOT the length variable which is for arrays):

```
String str = “spring is here”;
System.out.println(“String  str has the length: ” +
                        str.length());
```

- **length()** method will return the number of characters in the String str (14 characters in the above example, including spaces, since they are also characters… remember the ASCII table from the previous semester).

Strings – specific characters

- You can find out a particular character of a String using **charAt(int index)** which returns the character at the specified index/position.

  String str = "spring is here";

  char a = str.charAt(0); //s

  char b = str.charAt(7); //i

  char c = str.char at(14);

  //the above statement throws a StringIndexOutOfBoundsException

- **Note1:** Indexing starts at zero;
- **Note2**: the last character is at str.length()-1 which is **14-1** (or **13**)
- **Note3:** length is 14 (not 13) … like arrays

Strings – character indexes

indexOf() – takes in a char (or a String), and searches for it; if it is found, it returns the first matching position, if not, it returns -1; can also set the starting position; several overloaded versions;

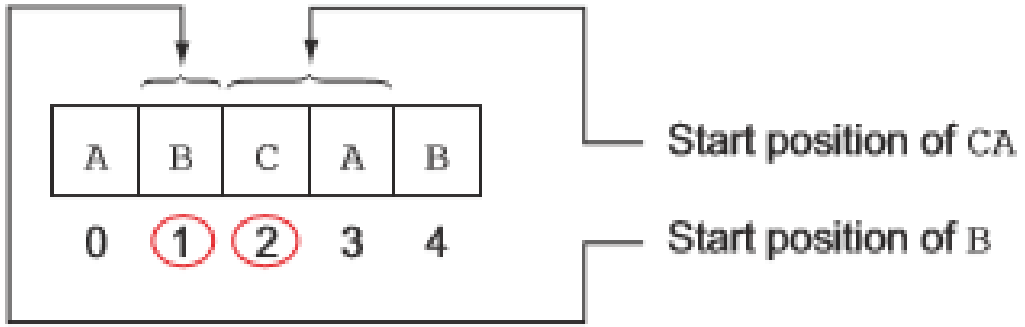
String letters = “ABCAB”;

System.out.println(letters.indexOf(‘B’)); //prints 1

System.out.println(letters.indexOf(“K”)); //prints -1 because there is no k in letters

System.out.println(letters.indexOf(“CA”)); //prints 2

System.out.println(letters.indexOf(‘B’, 2)); //prints 4



Strings – concatenation

- Concatenation means to put things together using the **+** operator.
- We can concatenate Strings together:

  String s1 = “hello ” + “world!”;
- We can also concatenate Strings to any primitive data types; for example:

  String s2 = “I am integer ” + 7;

  String s3 = “I am the boolean ” + true;

  String s4 = “I am the double ” + 3.4;

  String s5 = “I am the char ” + ‘c’;
- We can also concatenate Strings to any reference data types; for example:

  String s6 = “I am a scanner obj. ” + **new Scanner(System.*in);***

  String s7 = “I am a person obj. ” + **new Person(*);***

NOTE: if you haven’t overridden the **toString** method, the className@hashcode will be printed

Strings – trim and replace

- We can get rid of leading and trailing white spaces (space, tabs, etc.) using **trim** method which returns the reference to a new String object:

```
String s1 = “       I am a string with lots of spaces                    ”;
System.out.println(s1.trim());
/ * prints ‘I am a string with lots of spaces ’*/
```

- We can replace characters/sequence of characters using the **replace** overloaded methods:

```
String s2 = “hello”;
System.out.println(s2.replace(‘e’, ‘a’)); //prints hallo
System.out.println(s2.replace(“lo”, “ium”)); //prints helium
System.out.println(s2.replace(‘h’, “j”));
//will not compile because you cannot replace a string with a char or vice versa
```

NOTE: cannot mix the arguments…won’t compile

Strings – startsWith and endsWith

- **startsWith()** and **endsWith() –** determines whether the given String starts/ends with the specified character sequence which is also a String, so it returns a Boolean value:

```
String letters = “ABCBA”;
System.out.println(letters.startsWith(“AB”));//prints true
System.out.println(letters.endsWith(“A”));
//prints true
System.out.println(letters.startsWith(“CA”));//prints false
System.out.println(letters.endsWith(“CA”));
//prints false
```

Strings – substring

- **Substring method –** there are 2 overloaded versions:

1. ing:One returns a substring of the original string from the specified position to the end of the str

```
String name = “Pauline”;
System.out.println(name.substring(2)); //prints uline
System.out.println(name.substring(4));
//prints ine
```

2. The other also specifies the end position (not including the character at end position):

```
System.out.println(name.substring(2, 5)); //prints uli; note that character at index 5 is not included
System.out.println(name.substring(1, 2));
//prints a
```

Strings – Tokenisation

- String class also supports tokenisation process – where a long string (typically a sentence or a text) is split into its component tokens/words;
- **Tokens** are typically delimited by spaces;
- **split** – a method that returns an array of String object references which are the tokens of the original String (sentence/text);

```
String sentence = “Fundamentals of Programming 2”;

String[] tokens = sentence.split(“ ”); //split the sentence/text using a space

/* or we can use the pattern defined for java white spaces, and it will include all of them:
\p{javaWhitespace} → String[] tokens = sentence.split(“\\p{javaWhitespace}” ); */

for(int k = 0; k < tokens.length; k++){ //go through each element of the array

        System.out.println(tokens[k]);

}
```

OUTPUT: all the constituent words on a different line with no spaces

Note: you can split a piece of text/sentence using other patterns, such as punctuation, etc.

Strings – Case Conversion

- String class also supports conversion from one case to the other:

1. **toUpperCase** method returns a String object that has all the letters capitalised

```
System.out.println("hello1".toUpperCase());
/* prints HELLO1 */
```

2. toLowerCase method returns a String object that has all the letters in lower case

```
System.out.println("ANNA".toLowerCase());
/* prints anna */
```

Exercises

- Consider the following code:

String s = “I love Java ”;

What is the value of s.substring(3)?

“ove Java “

What is the value of s.indexOf(“j”)?

-1

What is the value of s.charAt(1)?

a single space

What is the value of s.trim()?

“I love Java”

What is the value of s now?

“I love Java ” – same: a string cannot be modified

- Consider the following code:

String s = “Easter!”;

What is the value of s.substring(s.length()-1)?

“!”

What is the value of s.startsWith(“e”)?

false

What is the value of s.replace(‘!’, ‘ chocolate!’)?

Easter chocolate!

s = s.replace(‘e’, ‘eee’); What is the value of s ?

Easteeer! – this time we made s point to a different string that was created by replacing e with 3 e’s.