

Classes and Methods

Objectives

- Apply OOPs concept when writing code in Java,
- Use eclipse tool to write good code,
- Apply arrays and strings in different programming scenarios,

Table of Content

Encapsulation through access Specifiers	Arrays
Getter(Accessor) and Setter (Mutators)	Initializing arrays
Creating and accessing attributes using	Enhanced for loop statement
Constructor	ArrayList Collections
Memory allocations	Multidimensional arrays
null	Command Line Arguments
Local vs Class declarations for an reference	String class
this	Immutability
instanceof	String comparison when created using constructors
Destruction	
Types of members	
Accessing static members	
Class constants	

Recall

- What is a Class?
- What is an Object?
- What is encapsulation?

Encapsulation through access Specifiers

- **public** access modifier can be used with class declarations and member declaration.
- **private** access modifier can be used only with inner class declarations and all member declarations (variable and methods).

Name

Registration number

Name of the degree

Current Semester

Student



What will you declare as private among the above attributes?

Generally a class is responsible for the integrity of its data. Therefore exposing the data within the class may prove to be dangerous.

But if you make all data private, how can other classes use them?

Getter(Accessor) and Setter (Mutators)

```
public class Student{  
    private String name;  
    private int regNo;  
  
    private String degreeName;  
    private int currentSemester;  
  
    public String getName()  
    public void setName(String nm)  
  
    public int getRegNo()  
    regNo cannot be set by any other class  
  
    public String  
    getDegreeName()  
    public void  
    setDegreeName(String dnm)  
  
    public int  
    getCurrentSemester()  
    public void  
    setCurrentSemester(int i)
```

Instance variables of Student class

Instance methods of Student class

Activity

- Complete the Student code in the previous slide. Add Logic to check the sanity of the values before assigning to the attribute.
- Did you notice the way the methods are named?
- Java's recommendation is to use these naming conventions.

Back to Conventions

- Writing code that follows conventions makes it easy for others to identify the constructs in your code.
- Local variable names must begin with _____ case.
- Constants' names must begin with _____ case.
- Class name must begin with uppercase and each subsequent word with an upper case letter. Names must be meaningful.
- Methods must be named like variables: name must begin with lower case and each subsequent word beginning with an upper case letter.
- In other words, members of the class are to be named in lower case (unless they are constants).
- Method names are generally verbs and variable names are nouns.
- Class names, variable names and method names syntactically can be same, but it is better to give different names where ever possible to avoid confusion.

Creating and accessing attributes using .

The **new** keyword is used to create objects.

```
Student s= new Student();
```

Access any member of a class, use . operator

```
s.name or s.getCurrentSemester();
```

② *The code listed below prints **null**, why?*

```
public class Test{  
    public static void main(String str[]){  
        Student s= new Student();  
        System.out.println(s.getName());  
    }  
}
```

Constructor

- The constructor is a special method for every class that helps initialize the object members at the time of creation.
- It is a special method because
 - it has the same name as the class name
 - does not have a return type.
 - Can be called only using '**new**' keyword when object gets created.
- There can be more than one constructor for a class.
- Space is allocated for an object only when the constructor is called. Declaring a variable of class and not calling new does not consume any memory!

Constructor Example

```
public class Student{  
    private String name;  
    private int regNo;  
    private String degreeName;  
    private int currentSemester;  
  
    /*Constructor 1 for student who has decided the degree  
    he/she is going to enroll into */  
  
    public Student(String nm, String d){  
        setName (nm) ;  
  
        regNo=generateRegno () ;  
  
        setDegreeName (d) ;  
  
        setCurrentSemester (1) ; }  
}
```

```
/*Constructor 2 for student who has not decided the  
degree he/she is going to enroll into */  
  
public Student(String nm) {  
    setName (nm) ;  
    regNo=generateRegno () ;  
    setCurrentSemester (1) ;}  
  
private int generateRegno () {  
    int nextNo=1;  
  
    //logic to generate regNo will be written later  
    return nextNo;}  
  
    // add setter and getters  
}
```

Creating objects by calling constructors

```
public class StudentTest{  
    public static void main(String args[]){  
        //Creating object using constructor 1  
        Student student1=  
            new Student("John", "M.C.A.");  
        //Creating object using constructor 2  
        Student student2= new Student("Mary");  
    }  
}
```

Student student1= new Student();

will result in a compilation error. This is because of the absence of no-argument or default constructor.

No argument constructor

- `new Student()` will try to invoke a constructor similar to the one below

```
class Student{  
    public Student(){..} }  
}
```

- An error will be generated because a no-argument constructor is not defined.
- When no constructors are defined for a class, compiler automatically inserts a constructor that takes no argument

```
public class Teacher{  
    public String name; Compiler inserts  
}
```

Compiler generated constructor

```
public Teacher()  
{  
    super();  
}
```

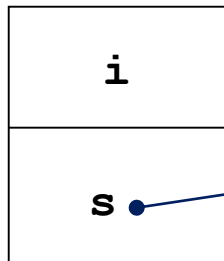
Learn about this in inheritance

Memory allocations

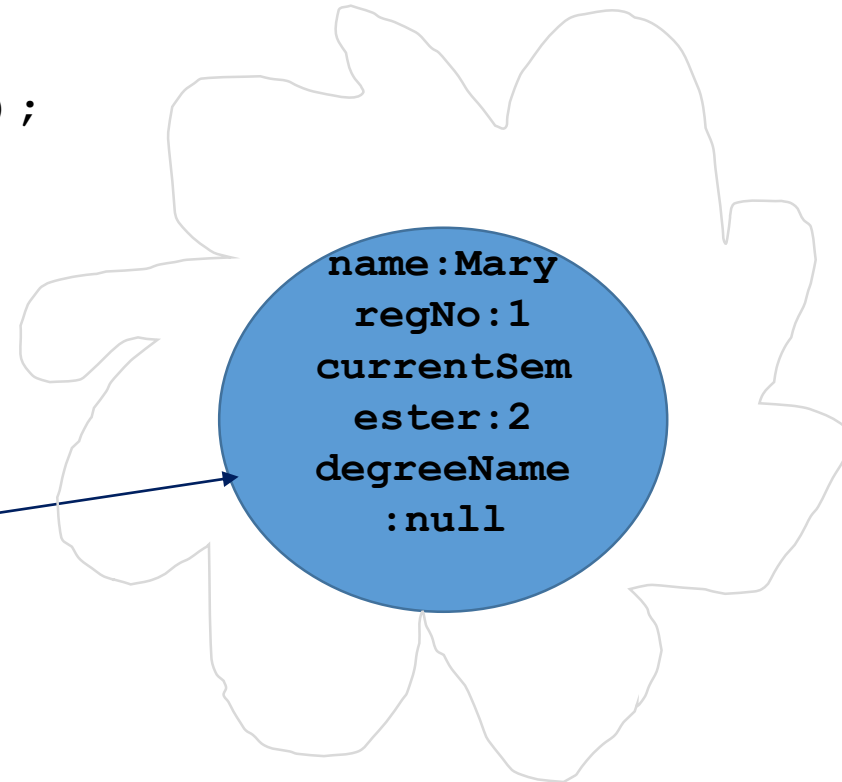
- Stack – for Local variables
- Heap – for references

```
public void test(){  
    int i;  
    Student s=new Student("Mary");  
    s.setCurrentSemester(2);  
}
```

STACK



HEAP



Activity: Multiple references to an object

```
Student student1= new Student("Mary");
```

```
Student student2=student1;
```

```
student1.setName("Merry Mary");
```

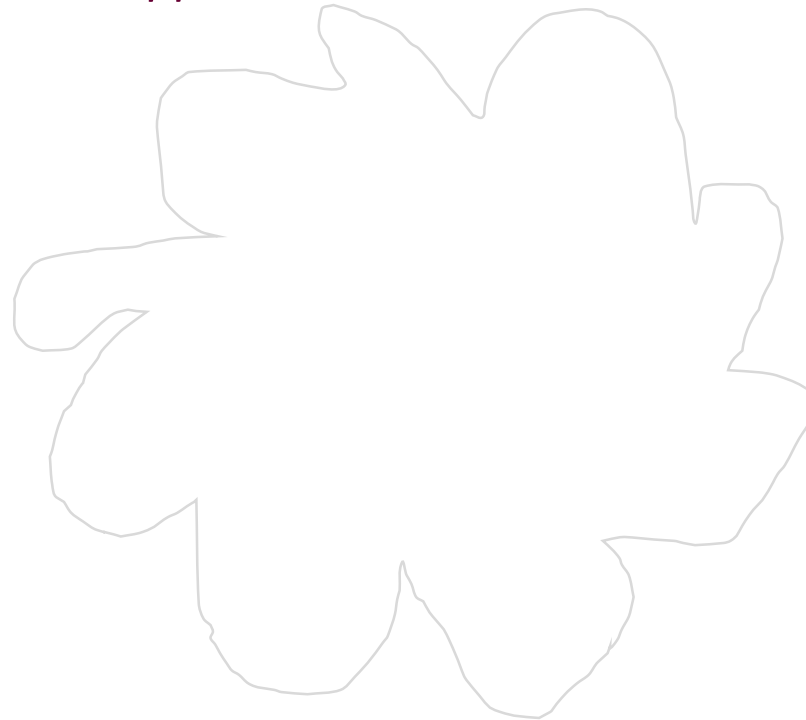
```
System.out.println(student2.getName());
```

HEAP



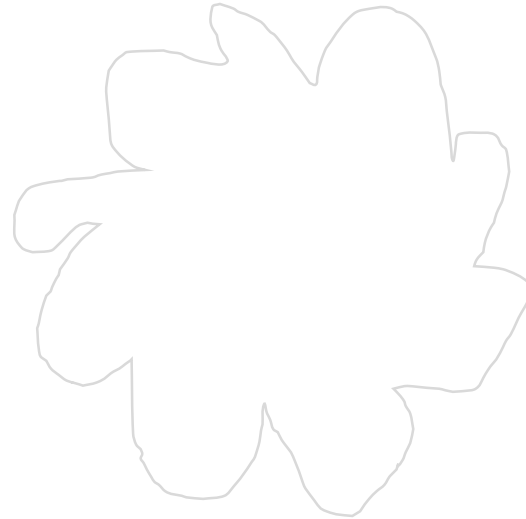
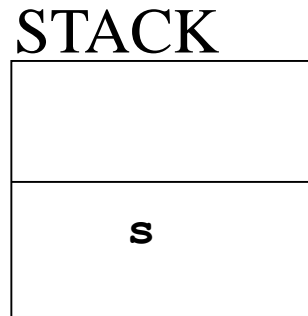
Complete the diagram. What will the code snippet print?

STACK



null

- Just declaration of a variable for a class does not create an object.
- **Student s;**



- **s** reference points to nothing or **null**.
- Default value of an object reference is **null**.

Local vs Class declarations for an reference

```
public class Test{  
    Student student;  
    public void test(){  
        student.display();  
    }  
}
```

On executing the code above, an error occurs at runtime
java.lang.NullPointerException

```
public class Test{  
    public void test(){  
        Student student;  
        student.display();  
    }  
}
```

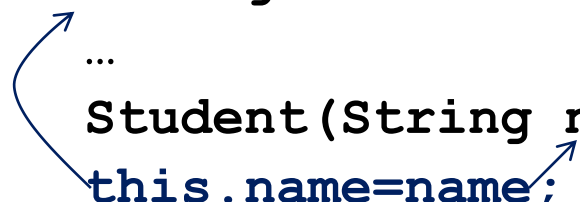
On executing the code above , a compile time error occurs.



Can you figure out why we get compile time error?

this

- **this** is a keyword used only inside a constructor or instance method and is used to refer to the current object.
- **this** is like a hidden reference that compiler provides to refer to current object.
- From the programmer's point of view **this** comes handy in two places
- To distinguish between local and class variables when they are the same.

```
public class Student{  
    String name;  
    ...  
    Student(String name) {  
        this.name=name;  
    ...  
    }}  

```

2. To call a constructor from another constructor of the same class

```
public class Student{  
    String name;  
    ...  
    public Student(String  
        name, String d){  
        this(name) ; calls  
        setDegreeName (d) ;  
    }  
}
```

```
public Student(String name) {  
    setName (name) ;  
    regNo=generateRegno () ;  
    setCurrentSemester (1) ;  
}  
...  
}
```

instanceof

- **instanceof** is an operator that is used to check if the instance is a type of class.
- Usage:
object-ref instanceof class-name
returns a **boolean** value.

Example:

```
Student s1= new Student("Mary");  
System.out.println(s1 instanceof  
    Student); // true  
System.out.println(s1 instanceof  
    College); //compilation error!  
System.out.println(null instanceof  
    Student); //false
```

Destruction

② *Any thing created in heap space must be freed. Why?*

- In Java, objects are automatically freed by a background thread called Garbage Collector.
- Garbage Collector frees the space
 - if the object reference is set to null and no other object reference refers to that object OR
 - if the object goes out of scope and its reference is not assigned to any other variable outside its scope.
- One cannot really determine when an object will be garbage collected.
- However, to explicitly invoke Garbage Collector following could be used

`System.gc()` or `Runtime.getRuntime().gc();`

Test your understanding

```
Student student1= new Student("Mary");  
Student studentref=student1;  
student1=null;
```



How many objects are created?

Will the student object created in the first line be garbage collected?

Types of members

- Instance Member

- Members of a class that is called using instance.
- They are part of every object.

- Class Member

- Members of a class that is called using class
- They are shared by objects of same class.
- Are created using the **static** modifier.
- Are initialized even before the instance variables are initialized and can be accessed even if the objects are not created.
- A **static** method can access only **static** variables. But an instance method/constructor can access both static and instance variables.

Example: Class Member

```
public class Student{  
    private static int gRegNo;  
    Student(String nm) {  
        regNo=generateRegno () ; ➔ constructor accessing static member  
        ...  
    }  
    private static int generateRegno () {  
        gRegNo++; ➔ static method accessing static member  
        regNo=gRegNo; ➔ static method cannot access instance member  
        return gRegNo;  
    }  
    public static int getGRegNo () {  
        return gRegNo;}  
    ...  
}
```

Accessing static members

- Static members can be accessed in one of the two ways:
 - *objectRef.staticvariable* or *objectRef.staticFunction()*
 - *ClassName.staticvariable* or *ClassName.staticFunction()*

```
public class StudentTest() {  
    public static void main(String args[]) {  
        Student student1= new Student("Mary");  
        Student student2;  
        System.out.println( Student.getGRegNo() );  
        System.out.println( student1.getGRegNo() );  
        System.out.println( student2.getGRegNo() );  
    }  
}
```

Class constants

- **final** keyword makes a variable declaration constant.
- A **final** member of a class either has to be initialized where it is declared or must be initialized in all the constructors of the class.
- If **static** modifier is also included, then the variable becomes class constant.

```
public class Student{  
    public static final int MAX_STUDENTS=3000;  
    ..  
}
```

*Note that while **local** variable can be made final, it cannot be made **static**!*

Tell me how?

- In Java there is no global scope.
- Then, how are the static members stored?
 - JVM Memory is actually divided into 3 sections
 - the Stack
 - the Heap
 - the Method Area
 - The Method Area has information about classes that is part of the executing code-
 - information about methods
 - information about static variables.
 - the Constant Pool for both strings and literals

Activity

- Analyse **main()** method's modifiers. Can main access instance and static members of its class?

Exercise

Create a class called Marks with the following details: regNo, MarksInEng, MarksInMaths and MarksInScience. Write getters and setters for the all variables. Use the class Student which is already created for getting the details of the student. Create a class called Standard with 5 students' details and write separate method for each of the following tasks and invoke the same.

- a) To display the entire reg no and the name of the students in the class in the ascending order of reg no.*
- b) To display the reg no and the name of the student who has got the highest percentage.*
- c) To display the reg no and the name of the student who scored highest mark in mathematics.*
- d) To display the reg no and the name of the student in the ascending order of the total marks in mathematics and science alone.*
- e) To display the reg no, name, total marks, percentage and rank of all the students in the descending order of rank.*

Arrays

- Arrays in java are like objects; they are created in heap.
- To create an array **new** keyword has to be used.

```
int sum=0,mul=0;  
int num[]= new int[5];  
  
for(int i=0;i<num.length;i++){  
    sum=sum+num[i];  
    mul=mul*num[i];  
}
```

Or `int [] num`

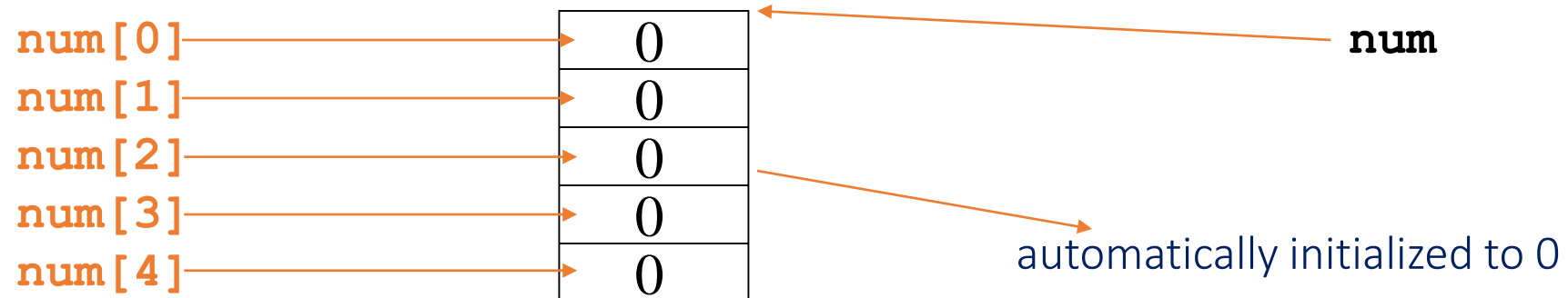
Allocating 5 spaces for int



Declaring array of type int

Returns the size of the array

Accessing array elements

```
int num[]= new int[5]; or int[] num= new int[5];
```



```
public class ArrayTest{  
    static int num[];  
    public static void main(String args[]){  
        System.out.println(num);  prints null  
        System.out.println(num[0]);  
    }  
}  error at runtime → NullPointerException
```


Initializing arrays

- `int [] a=new int[] {1,2,4,8,26};`

Or simply

`int [] a= {1,2,4,8,26};`

- The subscript can appear either before or after the variable name.
- The first syntax comes handy when we need to send an anonymous array to a method call

`method(new int[] {1,2,4,8,26});`

Activity

- `int a[] = new int[0];`
- `int a[] = null;`

What is the difference between the two statements?



Enhanced for loop statement

- Convenient way to iterate through arrays (and collection)
- Syntax:


`for(datatype variable: array) statement(s) ;`

Example 1:

```
int a[] = {1,2,3,4,5};  
for(int j:a)  
System.out.println(j) ;
```

Example 2:

```
Student s[] = new Student [2];  
                s[0] = new Student("Mary") ;  
                s[1] = new Student("John") ;  
for(Student s1:s)  
System.out.println(s1.getName()) ;
```



Exercise

- *Create a Stack class that can hold any number of integers. Provide methods like push and pop. Use appropriate OOPs principles.*
- *Hint: encapsulate an int array with some predefined size. During push operation, if there is an overflow, then the array size should be increased. Make sure that values are not lost in process of doing this.*

ArrayList Collections

- Arrays in java requires us to specify the size while it is created. In other words array does not grow dynamically.
- Java API provides `java.util.ArrayList` class that can be used in cases where we need dynamic array.
- The **`ArrayList`** class can hold only reference types.
- If we want **`ArrayList`** to be type-safe we can specify the type in the angular brackets.
- The enhanced for loop can also be used (apart from regular for loop) for iterating through it



*Locate **`ArrayList`** in Java API and go through the methods*

Example: ArrayList

```
import java.util.ArrayList;
class ArrayListTest{
public static void main(String[] s) {
ArrayList<Student> a= new
ArrayList<Student>();
a.add(new Student("Mary"));
a.add(new Student("John"));
a.add(new Student("Kate"));
for(Student o:a)
System.out.println(o.getName());
}}
```

Tell me how

- If **ArrayList** can hold only references, then how can we use it for primitives types?
- There are two ways to do this.
 1. Create your own class and encapsulate your preferred primitive type.
 2. Java API also provides corresponding class for every primitive type like **Byte, Short, Integer, Long, Float, Double, Boolean, Character**.

So instead of `int i=1;` we can use `Integer i=1;`

Exercise

- *Create an ArrayList that can hold integers.*
- *Use methods to insert, delete and find values in the ArrayList.*

Multidimensional arrays

- Declaration and creating
 - `int[][] m= new int[3][3];`
- Initializing
 - `int[][] matrix= new int[][]
{{0,0},{0,0,0},{0,0,1}};`
 - `int[][] matrix= {{0,0},{0,0,0},{0,0,1}};`
- Accessing
 - `System.out.println(matrix[0][0]);`
- Length
 - `matrix[0].length → 2`
 - `matrix.length → 3`

Exercise

Write a program to construct two matrices and display the sum of those.

Command Line Arguments

- Command line arguments are sent as string arrays through main function's argument.

```
public class Test{  
    public static void main(String args[]){  
        /*command line argument passed */  
        /* java Test Mary */  
        System.out.println(" length: "+args.length); //  
        Length=0  
        System.out.println(args[0]); // Mary  
    }  
}
```

Activity

- *Checkout what happens if you run Test with and without supplying command line arguments ?*
- *Why doesn't the first line end up in NullPointerException?*

String class

- **String** class has been part of Java right from its inception.
- In Java, strings can be created without using char array.
- The String class has convenient methods that allows working with strings.

Constructors:

```
String()  
String(String s)
```

Examples of creating String object:

```
String s="abc";  
String s= new String();  
String s= new String("Hello");  
String s1= new String(s);
```



*Open the Java doc and locate String API.
Go through the methods*

Tell me why?

- Why do we require equals() method to compare Strings ?
Can we not compare using ==?
- == works fine with primitive data types.
- But with references, (since they are like pointers) , == will actually compare the addresses.
- What we want here is to check equality of value of strings.

```
String s1= "abc";  
String s2=new String(s1);  
System.out.println(s1==s2); // returns false  
System.out.println(s1.equals(s2)); // returns true
```

Exercise

Write class that declares the following String.

“The quick brown fox jumps over the lazy dog”.

Perform the following modifications to the above string using appropriate Methods.

- a) Print the character at the 12th index.*
 - b) Check whether the String contains the word “is”.*
 - c) Add the string “and killed it” to the existing string.*
 - d) Check whether the String ends with the word “dogs”.*
 - e) Check whether the String is equal to “The quick brown Fox jumps over the lazy Dog”.*
 - f) Check whether the String is equal to “THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG”.*
- Cont...

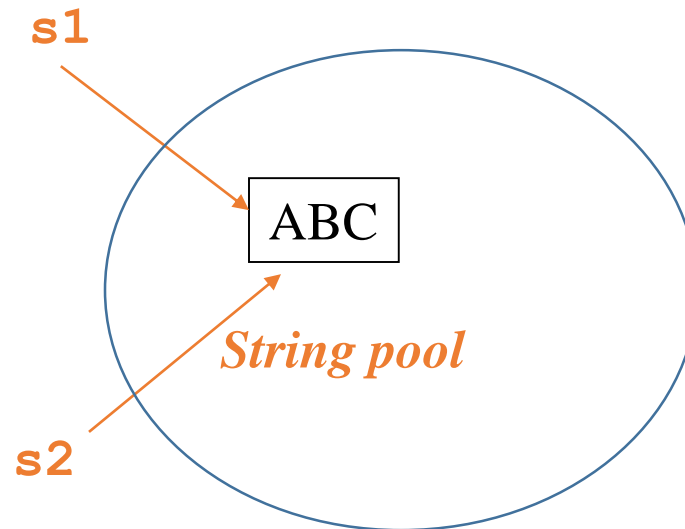
- g) Find the length of the String.*
- h) Check whether the String matches to “The quick brown Fox jumps over the lazy Dog”.*
- i) Replace the word “The” with the word “A”.*
- j) Split the above string into two such that two animal names do not come together.*
- k) Print the animal names alone separately from the above string.*
- l) Print the above string in completely lower case.*
- m) Print the above string in completely upper case.*
- n) Find the index position of the character “a”.*
- o) Find the last index position of the character “e”.*

(30 mins)

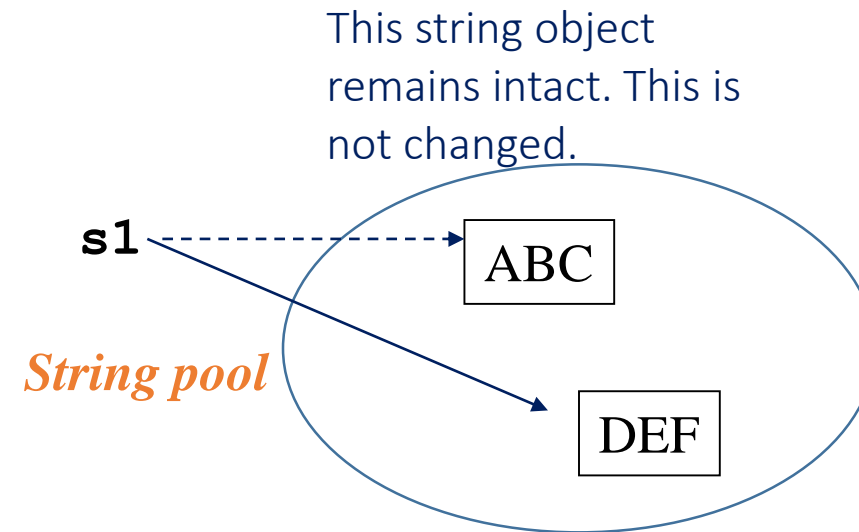
Immutability

- Strings are immutable in Java.
- String literals are very heavily used in applications and they also occupy a lot of memory.
- Therefore for efficient memory management, all the strings are created and kept by the JVM in a place called string pool (which is part of Method Area).
- Garbage collector does not come into string pool.

```
String s1="ABC";  
String s2="ABC";
```

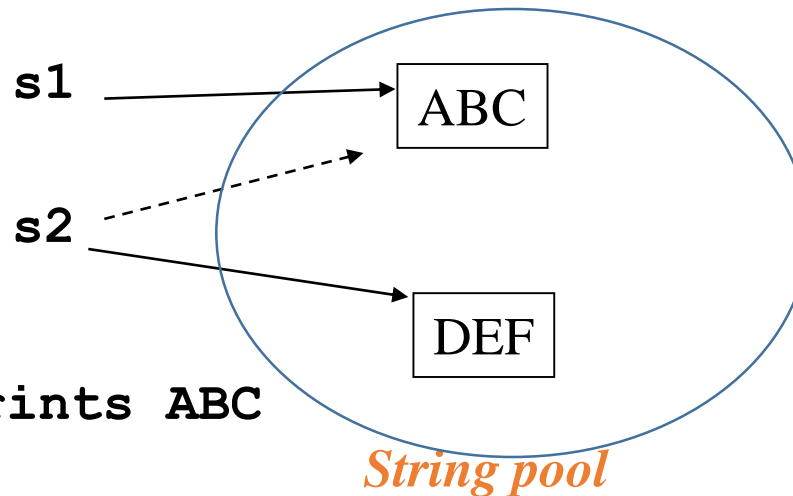


```
String s1="ABC";  
s1="DEF"
```



Assigning string references:

```
String s1="ABC";  
String s2=s1;  
s2="DEF";  
System.out.println(s1); // prints ABC
```



Test your understanding

What will the code print?

```
String s1="ABC";  
String s2=s1;  
System.out.println(s1==s2);
```

```
s2="DEF";  
System.out.println(s1==s2);
```

```
String s4="ABC";  
System.out.println(s1==s4);
```

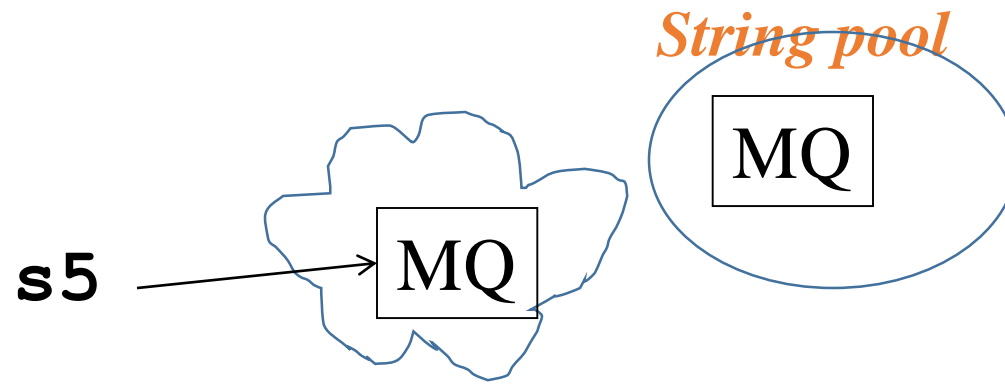
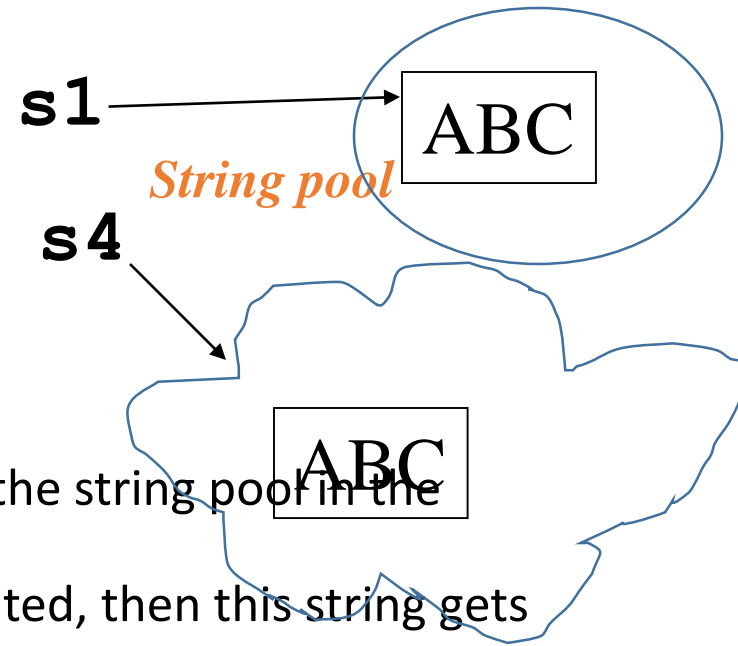
```
String student1= "Mary";  
String studentref="John";  
String studentref=student1;  
student1=null;
```

How many instances of String will be eligible for garbage collection?

String comparison when created using constructors

```
String s1="ABC";  
String s4=new String("ABC");  
System.out.println(s1==s4);  
//Prints : false
```

- The String constructor creates the string outside the string pool in the heap.
- If there is no string represented by the string created, then this string gets added to the pool also.
- **String s5=new String("MQ");**



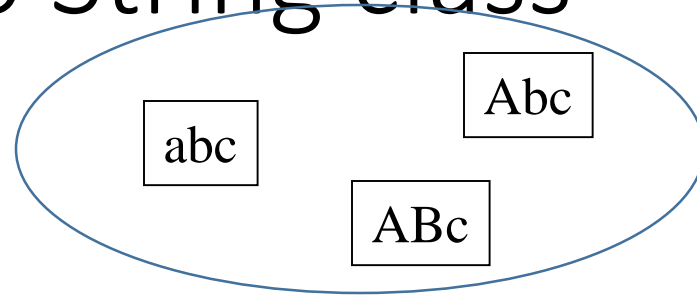
Test your understanding

```
String s1= "abc";  
s1=s1.replace('a', 'A');  
s1.replace('b', 'B');  
System.out.print(s1);
```

What will the code print?

How many Strings are created in the pool?

Alternate to String class



- In the example in the previous slide, 3 strings were created in the pool. What was needed after the 3rd line was the last string ABc.
- Since garbage collector does not go to the string pool, we have to be very careful when we work with the strings.
- If there are lots of String manipulations in your application and you are concerned about memory, then String class should not be used.
- JSE comes with another class for such operations – **StringBuilder** and **StringBuffer** class.

Exercise

- *Given an array of 10 students' names. Convert the array as a single string and print it.*

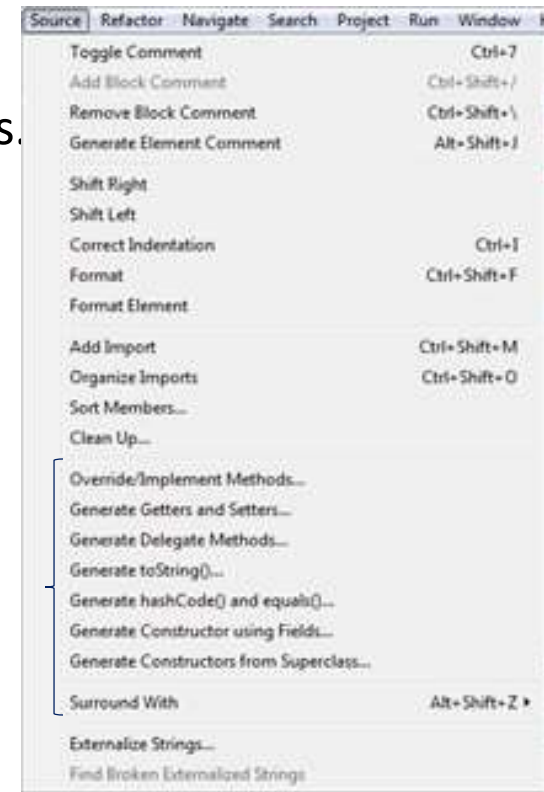
Hint: Efficient use of memory is the focus here

Exercise

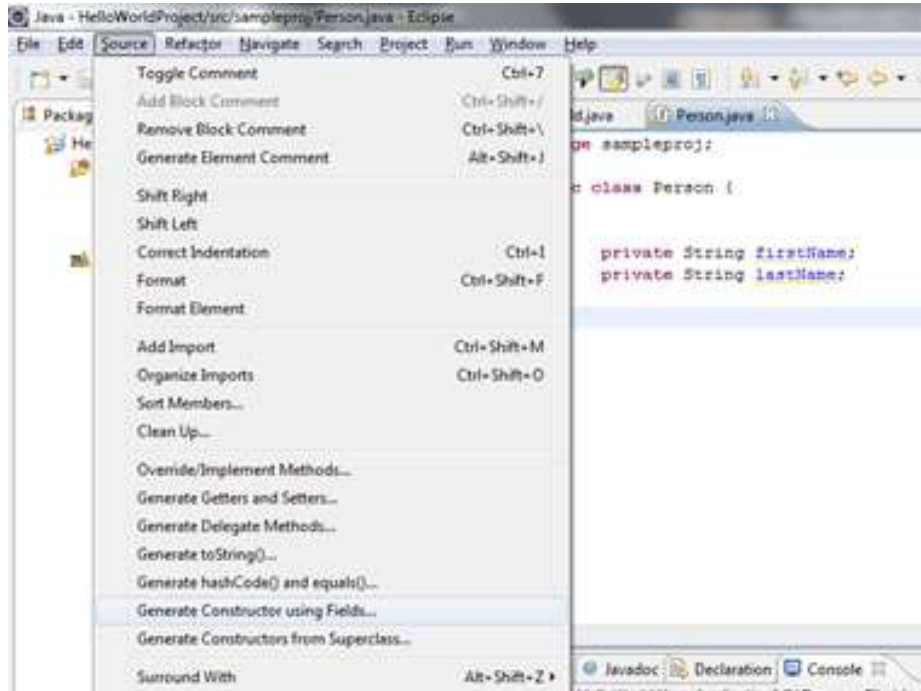
- *Write a program to check whether the given strings are an anagram or not. An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "Ate" is an anagram of "Eat". The program should ignore white space and punctuation.*

Generating Code in Eclipse

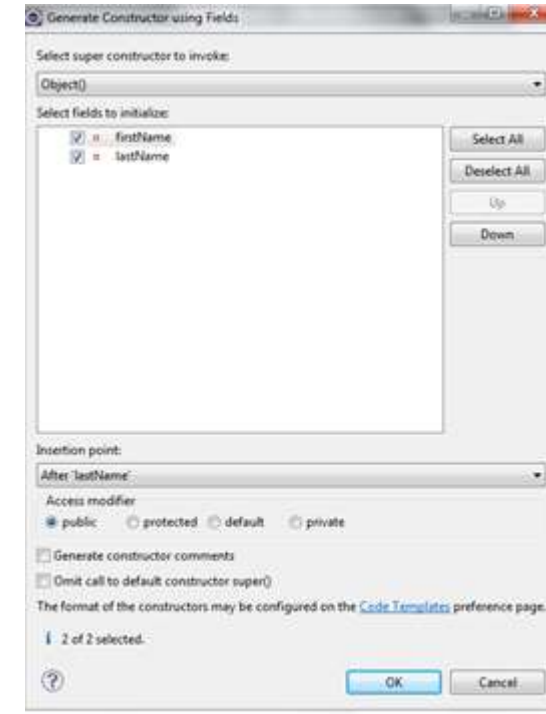
- Eclipse provides the user many ways to generate code automatically by which the time spent on the development can be saved.
 - Can generate getter and setter methods for attributes.
 - Can generate constructors.
 - Can override/implement methods from superclasses.
 - Can generate the toString(), hashCode() and equals() methods.
 - Can generate delegate methods.
- Code generating options can be found in the Source menu.



To insert the Constructor using fields, Select
Source → Generate Constructor from Fields



Select Ok



The code is generated.
Similarly all the options
can be used.

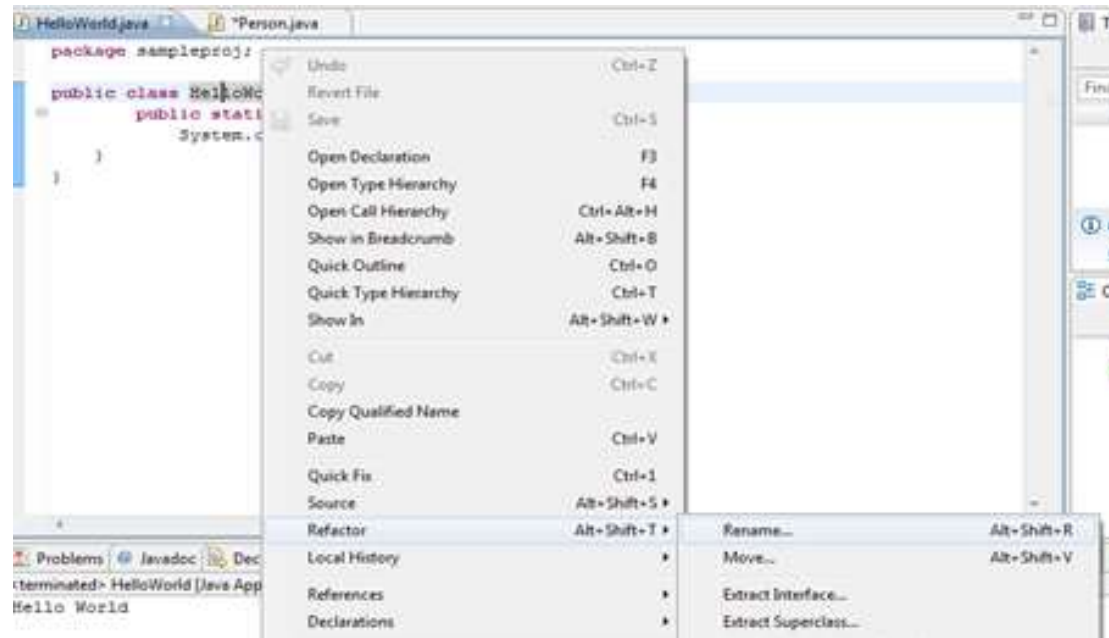


Activity: Code Refactoring

- There is an option in eclipse, where we can modify the code without changing its behavior as per our need. For example if we need to rename a Java class or method, we can do this in eclipse which will update in all the references of that class or method. This process is called code refactoring.
- Eclipse supports simple refactoring activities, such as renaming or moving.

For Example, to rename your class or method, right click on it and select Refractor→Rename.

Eclipse will make sure that all calls in the Workspace to this class or method will also be renamed.



Summary

- The new keyword is used to create objects
- The constructor is a special method that helps initialize the object members at the time of creation. It has the same name as the class name.
- this is a keyword which is used to refer to the current object.
- Instanceof operator is used to check if the instance is a type of class.
- In Java, objects are automatically freed by a background thread called Garbage Collector.
- Arrays in java are like objects and they are created in heap.
- Strings are immutable in Java.