Classes and Methods

Quick Revision

- What is a Class?
- What is an Object?
- What is encapsulation and how will you achieve this in Java?

Public & Private Access Specifiers/modifiers

- **`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).

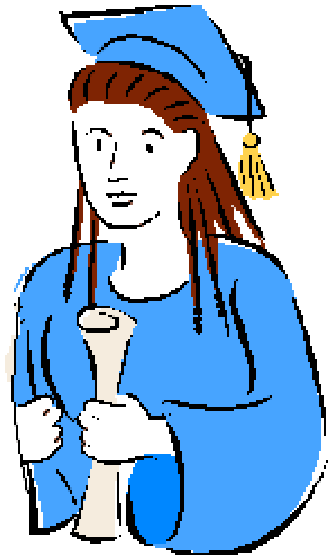
Name

Registration number

Name of the degree

Current Semester

Student



What will you declare as private among the above attributes?

If your answer is all of them, then it is right.
Generally a class is responsible for the integrity of its data. Therefore exposing the data within the class may prove to be dangerous.

Making attributes/variables private

```
public class Student{

private String name;

private int regNo;

private String degreeName;

private int currentSemester;

…

}
```

What is the use of a class if other classes cannot access any of its data?
For example, Exam class will want to know regNos of all students.
Result class will want to update the **`currentSemester`** *once the student passes.*

Accessors and Mutators

- Accessor
    - Getter method
    - Used to return the value of the attribute
    - Sometimes values may be same as the value of attribute, sometimes logic could be written to return the value expected by other classes. (Example coming up)
- Mutator
    - Setter method
    - Used to assign a value of the attribute
    - Logic to check the sanity of the value assigned to the attribute is written in the method

Example with getter and setter

```
public class Student{

private String name;

private int regNo;



private String degreeName;



private int currentSemester;
```

Instance variables of Student class

```
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 methods of Student class

Full Example

```
public class Student{
private String name;
private int regNo;
private String degreeName;
private int currentSemester;

public String getName(){return properCase(name);}

private String properCase(String nm){
// Logic to be written later
return nm;
}
public void setName(String nm){
if(nm==null)
System.out.println("Name cannot be null");
else
name=nm;
}
```

Values returned may not be same as attribute

Logic to make sure name is not null.

What more checks do you need to perform here to make sure names are valid

```
public int getRegNo(){ return regNo; }
public String getDegreeName(){ return degreeName; }
public void setDegreeName(String dnm){
if(dnm==null)
System.out.println("Degree name cannot be null");
else
degreeName=dnm;}

public int getCurrentSemester(){
return currentSemester;}
public void setCurrentSemester(int i){
if((i<0) || (i<currentSemester ))
System.out.println("Invalid value for current
semester");
else
currentSemester=i;
}}
```

Points to Note

- Take a look at the way the methods are named
- Getters
  - are named beginning with “get” followed by the name of the variable beginning with upper case
  - Has no arguments
  - Returns the value of type which usually same as that of the variable
- Setters
  - are named beginning with “set” followed by the name of the variable beginning with upper case
  - Has no return value (void)
  - Takes an argument of usually same type as that of the attribute

Back to Conventions

- Writing code that follows conventions makes it easy for others to identify 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 variables name 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.

Accessing attributes using .

How do you create a Student object?

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

Access any member of a class, use `.` operator

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

What will the code below print?

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

Is it correct to allow creation of Student objects without name. Also registration number needs to be auto-generated as and when student object is created.

Constructor

Need a way in which only valid Student objects are created.
Need to provide values for student object at the time of creation.
Need a 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.
- Only when constructor is called, space for it is allocated. Declaration of variable of class type does not consume any memory!

Constructor Example

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

A constructor has the same name as the class and has no return value.

Found a place where registration number can be assigned.

```
/*Constructors 2 for student who have not decided the
degree they are 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");
}
}
```

What will happen if we create a Student object like we created in session 1?

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

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 since there is no no-argument constructor defined
- *If we want Student objects to be created using* ***`new Student()`*** *then we must provide a constructor. (We do not want such creations for the reasons we already discussed)*
- *But then how did it work in the first session? We never wrote any constructor there!*

Compiler generated constructor

- When no constructors are defined for a class, compiler automatically inserts a constructor that takes no argument.

```
public  class Student{
public long regno;
public String name;

…
}
```

Compiler inserts

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

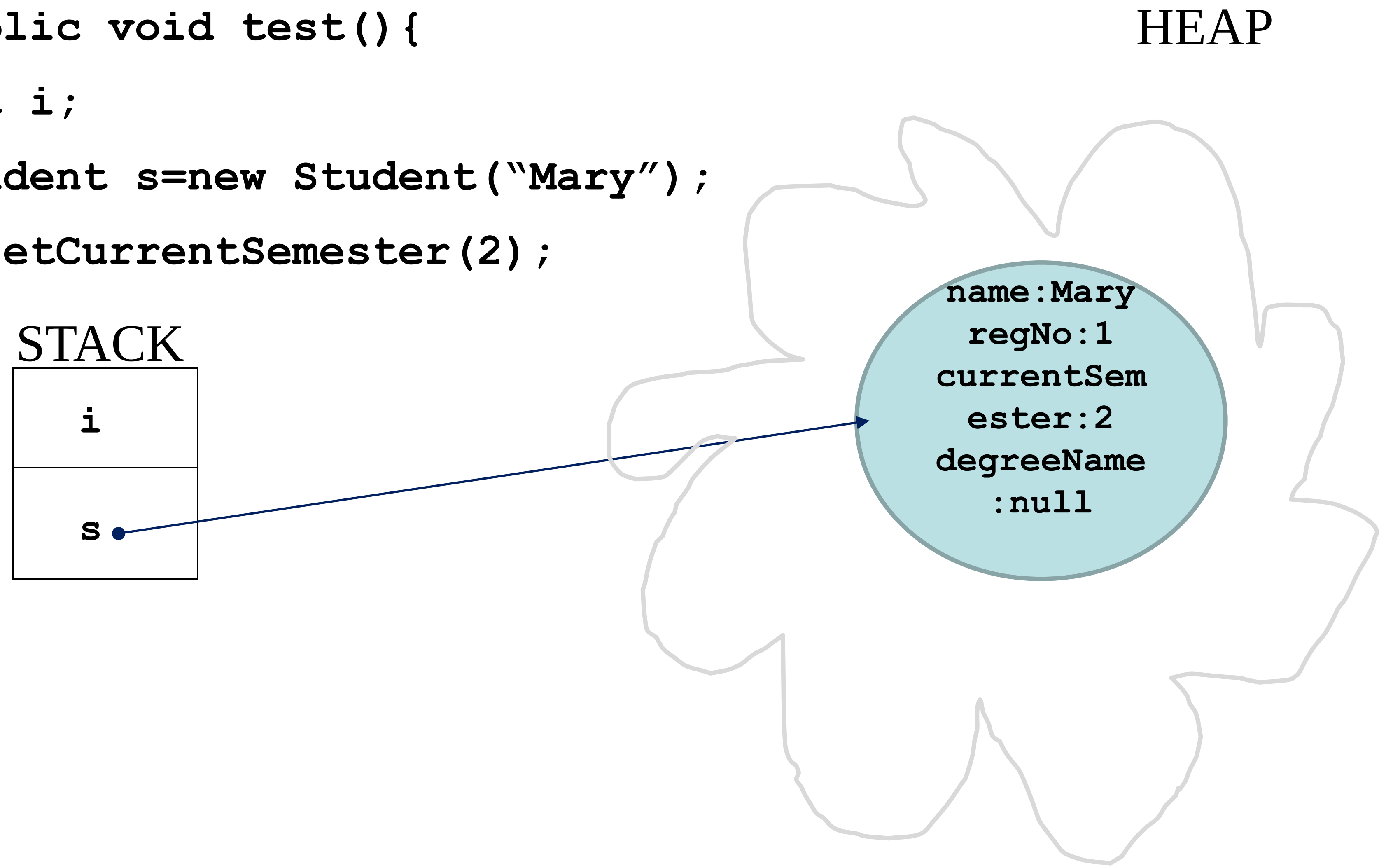
Learn about this in inheritance

Memory allocations

- Local variables are created in **Stack.**
- Objects are created are created in **Heap**. It is like allocating memory using **`malloc.`**
- What is returned back to the program on invoking **`new`** is a **reference** that is again in **Stack.**
- C++ programmers can compare this with the reference in C++. Sort of alias name for an variable.
- For C programmers, references is like constant pointer to a variable that is a pointer using which value of the variable can be changed but pointer itself cannot be changed to point to some other variable. The variable here is an object.
- In Java, all variables that created for objects are references.

Memory allocation by example:

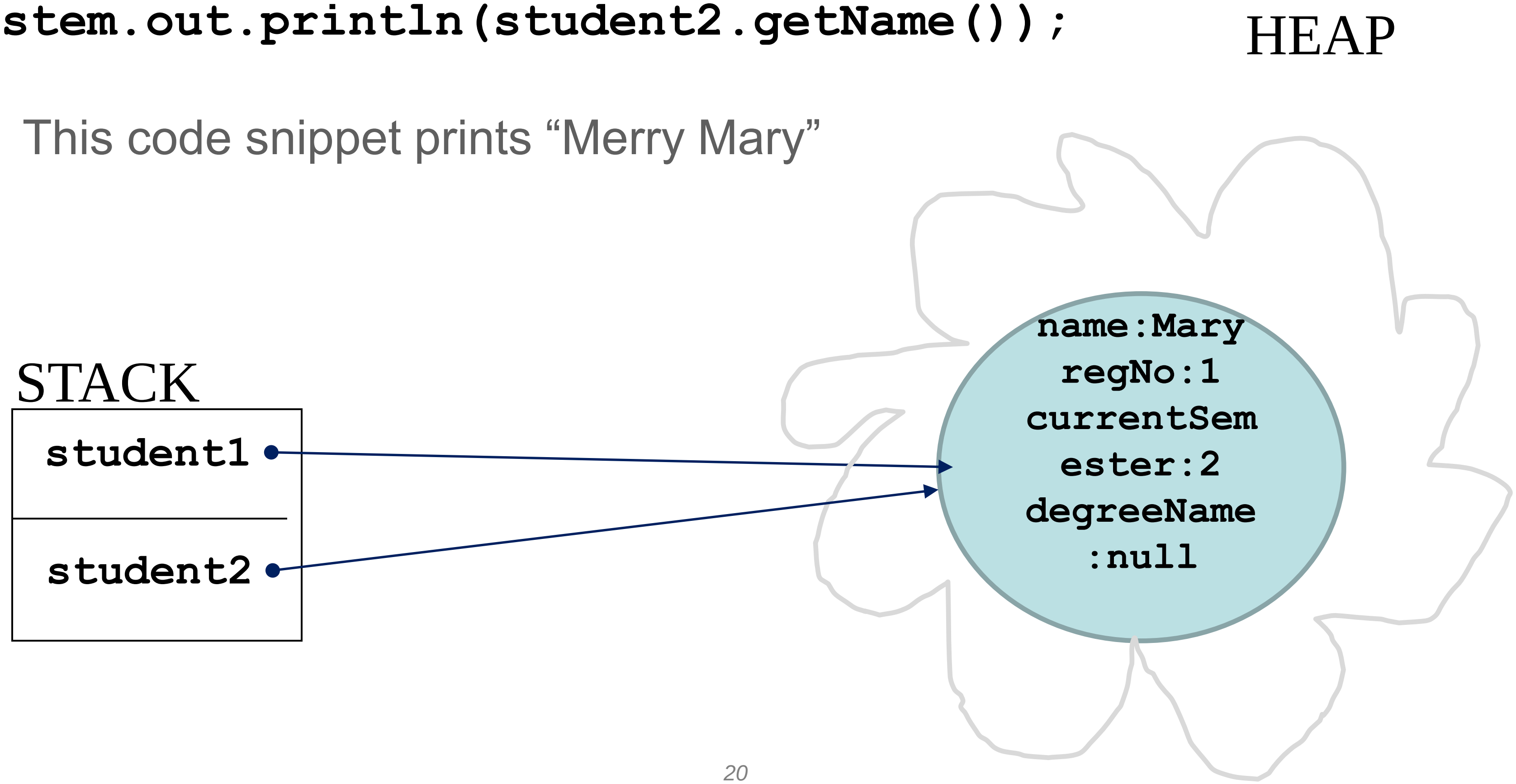
```
public void test(){
int i;
Student s=new Student("Mary");
s.setCurrentSemester(2);
}
```



Multiple references to an object

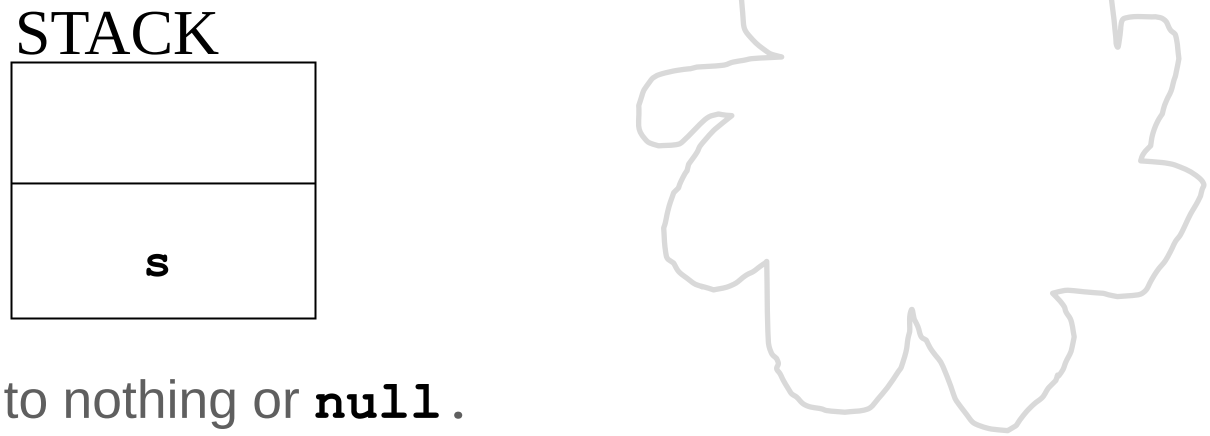
```
Student student1= new Student("Mary");
Student student2=student1;
student1.setName("Merry Mary");
System.out.println(student2.getName());
```

- This code snippet prints "Merry Mary"



`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`**.
- Based on the scope of the declaration (local or class) either compile-time or run-time error is generated on invoking a method on a reference which is **`null`** .
- Calling an instance method on a **`null`** object gives runtime error called **`NullPointerException`**

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 , an error occurs at compile time.

Do you recall something similar. Can you figure out why this is so?

Another point to note about scope in Java is that there is no global declarations!

Why this?

How can you invoke an instance method?

Through object reference, that is
`s.getName();`
But we called instance methods from constructor without any object?

```
public Student(String nm){

setName(nm);
…
}
```

Similarly we can call any instance method/ constructor from any other instance method/ constructors of the same class. Like **`getName()`** called instance method **`propercase(name)`**.

```
public String getName(){
return properCase(name);
}
```

On what object are we calling these methods?

Right, it all began from **`main()`** method.

- First **`student1`** was created – new invoked the constructor. student1 got created. So **`setName()`** was called on **`student1`** object.
- Second **`student2`** was created – new invoked the constructor. student2 got created. So **`setName()`** was called on **`student2`** object.

Call is made on the object which invoked the method in the first place.

Instance method and constructor call it as the current object or **`this.`**

Compiler converts the code as:

```
public Student(String nm){
this.setName(nm);
…
}
```

```
public String getName(){
return
this.properCase(name);
}
```

`this`

- **`this`** is a keyword used only inside an constructors 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 programmer's point of view **`this`** comes handy in two places
  - To distinguish between local and class variables in cases where they are same
  - To call a constructor from another constructor of the same class

this to distinguish between local and class variables

- In the example below, the name of the parameter for Student constructor and name of the attribute are same. Both are “name”.
- In cases where there is a name clash, local names are considered.

```
public class Student{
String name;
…
Student(String name){
this.name=name;
regNo=generateRegno();
currentSemester=1;        }}
```

Calling constructor from another constructor

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

calls

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

- When there are many constructors, pretty much most initialization statements are repeated in all constructors.
- Instead of this repetition, it would be better if common initialization could be put on one constructor and all constructors can call this constructor.
- That is when **`this()`** comes into picture.
- Calling constructors is possible either by using **`new`** or by using **`this()`**.
- **`this()`** can be called only from constructor.
- Call to **`this()`** must be the first statement in the constructor.

`instanceof`

- An instance is another name for object reference.
- Class is also a user-defined data type (like struct in C)
- **`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
```

At this point **`instanceof`** *is not of much use to us.*

The real benefit of it will be realized in inheritance section.

Destruction

- Objects are created in ___________.
- *Any thing created in heap space must be freed. Why? Why do you free memory that you allocate using* **`malloc`***?*
- Heap is a memory location that is available to the executing application. It is also called a free store. That is the reason why allocation of memory in heap is also called dynamic allocation.
- If memory is not released after use, at some point if dynamic allocations are more, there will be no space for further allocations. Hence it is recommended that memory is released after its use.
- In languages like C and C++, memory allocated must be freed explicitly by the programmer.
- In Java, objects are automatically freed by a background thread called Garbage Collector.

Garbage Collector

- Object in java are garbage collected .
- How will the Garbage Collector know when to free the object?
- 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.
- Garbage Collector is a low priority background thread.
- Therefore, one cannot really determine when an object will be garbage collected.
- However, programmers can explicitly invoke Garbage Collector by calling

`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?

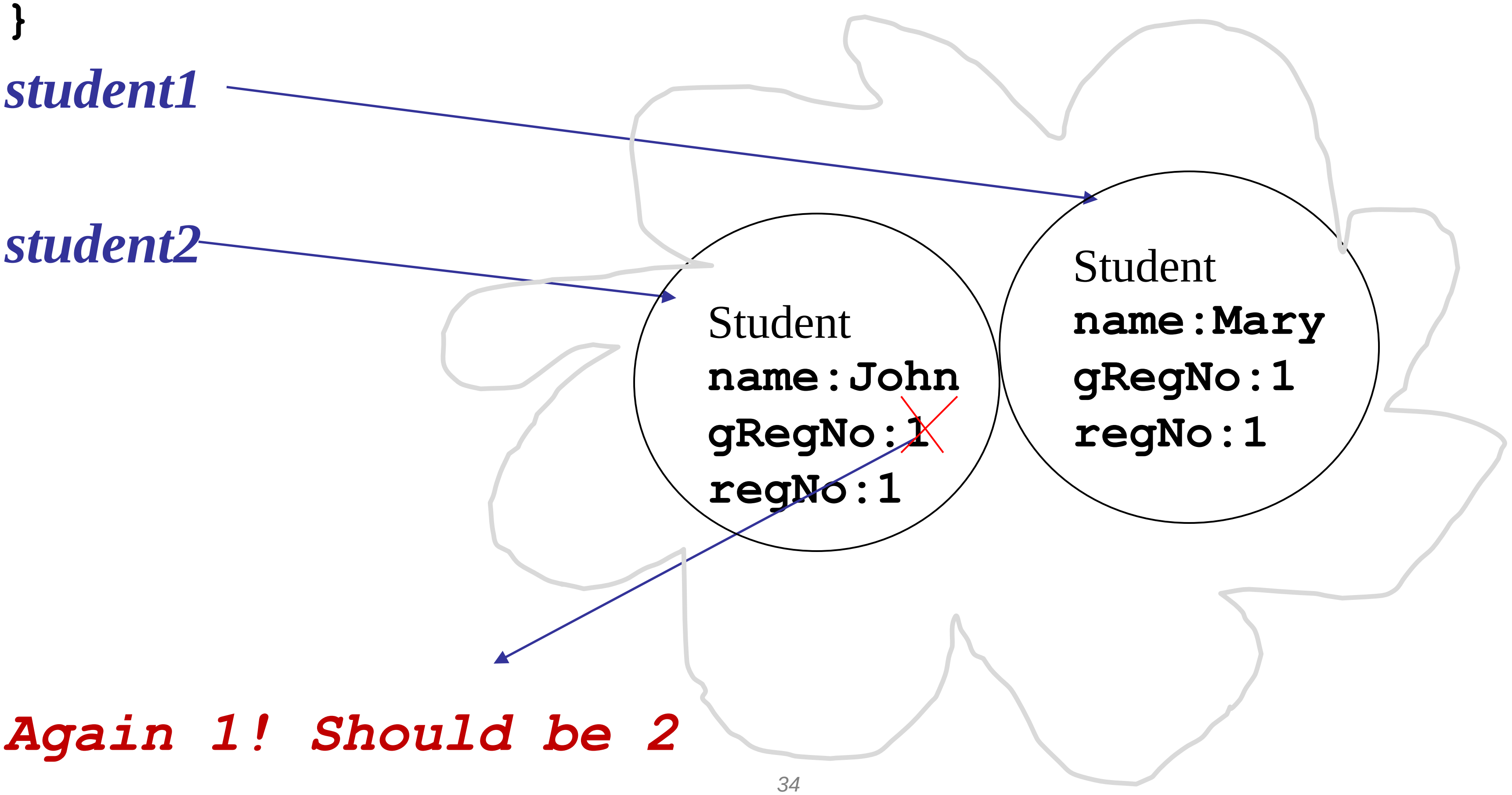
Ans: 1, No

Instance members – enough?

- One important part of the Student class is the generation of the registration number.
- The registration number for each student should be unique and continuous numbers must be assigned to students in the constructor when student objects are created.
- Now to give each new object a unique registration number, a variable must be maintain that can hold the value of the last number allocated to the student object.
- Let us try it with instance member.

```
public class Student{
private int gRegNo; //To store the last count of registration
number
private String name;
private int regNo;
…

public Student(String nm){
regNo=generateRegno();
setCurrentSemester(1);
}

// increments gRegNo each time it is called.
private int generateRegno(){
gRegNo++;
return gRegNo;}
…
}
```

```
public static void main(String str){
Student student1= new Student("Mary");
Student student2= new Student("John");
System.out.println(student2.getRegNo());
}
```



Again 1! Should be 2

Test Your Understanding

- How would you solve it in C if you had to assign a consecutive number to an array of struct variables?

- Since gRegNo is part of every object, our objective is not achieved. What is needed is something like global variable in C. Something that is not part of any object.
- You cannot have global variables in Java.
- Class variables help us do the same thing here.

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.

So far all the members that we have came across are _____________.

Class Members

- Class Members are the members that are not part of any object but part of the class.
- They are created using the **`static`** modifier.
- A **`static`** members of a class is global to all objects created by that class.
- All the objects of the class that declares the **`static`** variable can share the value of **`static`** variable.
- Changes in **`static`** variable are visible to all objects of the class. And any object of that class can change the static variable.
- Static variables 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.

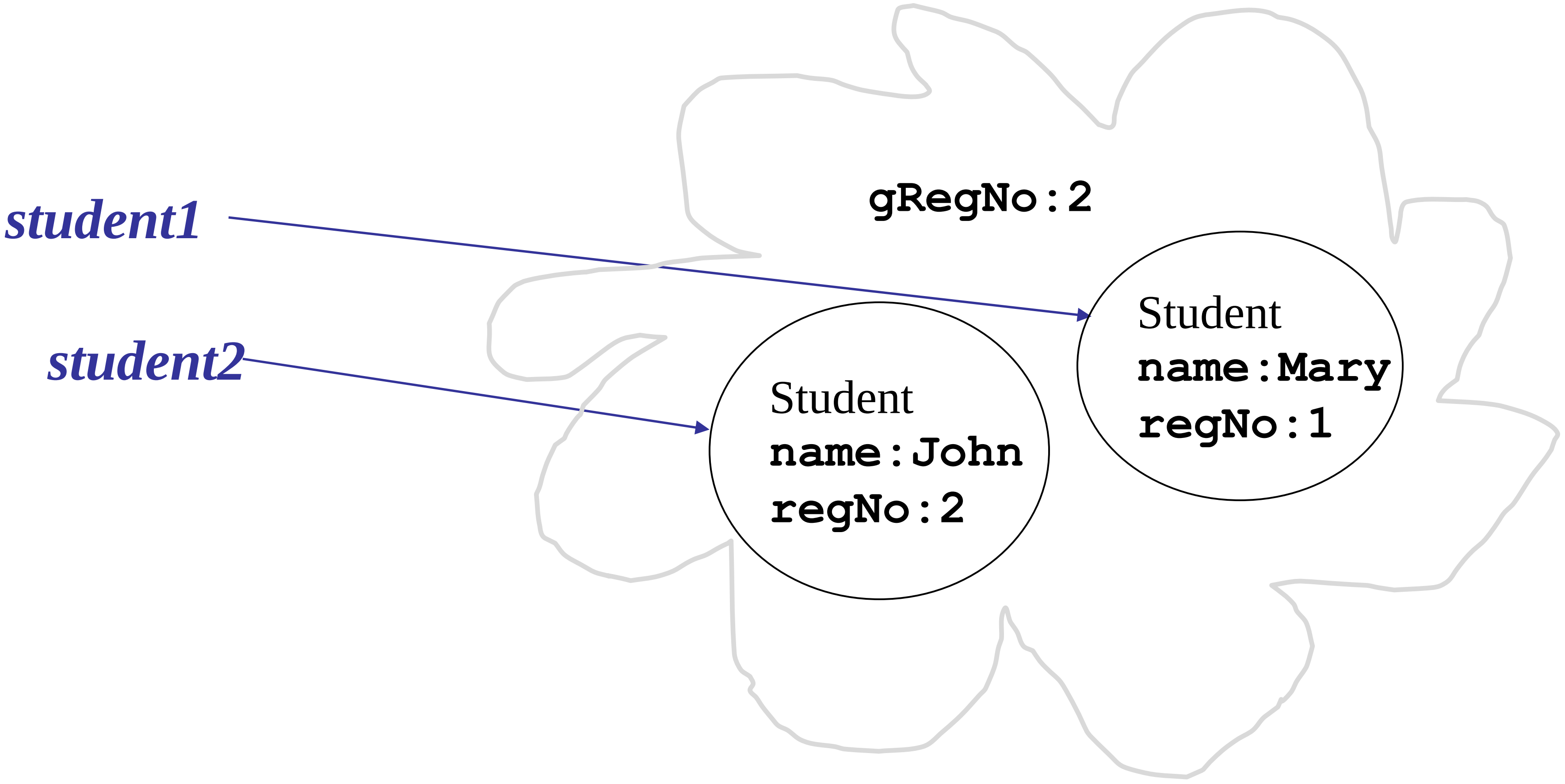
Example

```
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()***
  - This is not recommended practice because it does not make clear it look like they are class variables.
  - ***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() );
}}
```



student1
student2
gRegNo:2
Student
name:Mary
regNo:1
Student
name:John
regNo:2

Tell me how?

- In Java there is no global scope. There is only stack and heap. Then, how are 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

Tell me how?

- `main()` is a `static` method and is part of class. Then how is it that main can access instance members?

- A `static` method cannot access instance methods directly or using `this`. This is because,
  1. Instance methods can be accessed only through instances.
  2. `static` methods can be accessed even without an instance. Since instance variables are part of object, when `static` method accesses instance variable directly, there is a big question mark as to where is this instance variable.

- But if an instance is created inside a static method, using this instance, one can always call instance members.

```
public static void main(String args[]){
Student student1= new Student("Mary");
student1.setCurrentSemester(2);}
```

- If `main()` method is part of a class, then it does not need any instance to call any `static` method of that class.

Exploring modifiers of `main()`

`public static void main(String args[])`

- Can also be written as

  `static public void main(String args[])`

But **`public static`** is preferred

`main()` method is

  - is **`public so that JVM can access this method`**
  - **`static`** so JVM need not create an object of this class to call main method.

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`****!*

Arrays

- Array is a data structure that can hold a number of values of a data type.
- Arrays in java are like objects; they are created in heap.
- To create an array **`new`** keyword has to be used.
- In the example (in the next slide) **`num`** is a reference that points to an array of object created in the heap.
- Please note in the **`for`** loop a very handy method that is used to get the **`length`** of the array.
- Array elements are automatically initialized to the default value based on the type of the array.
- Creating **`int num[]`** does not create an array.
- Also **`int num[3]`** is wrong syntax.
- Subscript can be before or after variable declaration.

Arrays- creating and initializing

```
int sum=0,mul=0;
int num[]= new int[5];
```

Allocating 5 spaces for int

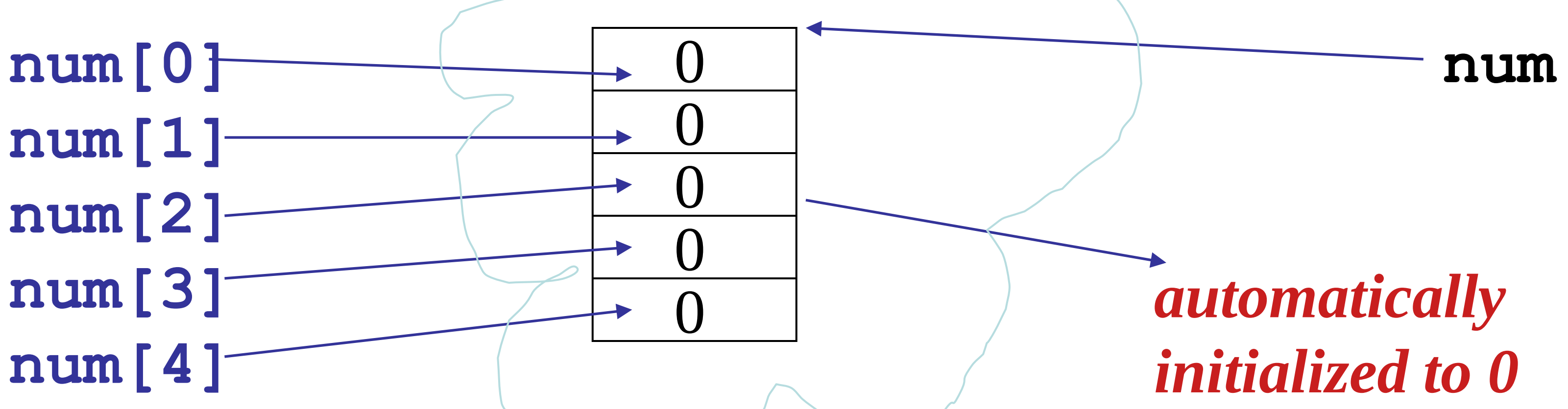
Declaring array of type int

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

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.
        Can you recall the exception that it throws?
```

- The member variable **num** in the above code is an object reference. Since the **num** object is not created, **num** is assigned a value **null**. This, as we have seen earlier, is what happens in case of object references.
- If we try to print an array element which is not created, an error occurs at run time.

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 pr 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});
```

Empty Array

- `int a[]= new int[0];`
- The above statement creates an array of length 0 indicating an empty array.
- This is different from array being **`null`**.
- There are many cases in which an "empty" object can be used instead of a **`null`** object reference.
- This is usually preferable since it helps eliminate one of the most common problems encountered during development -- the **`NullPointerException`**

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);
```

The variable in the **`for-each`** statement must be of same type as that of elements in the array

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());
```

Multidimensional arrays

- Declaration and creating
  - `int[][] m= new int[3][3];`
- Initlalizing
  - `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`

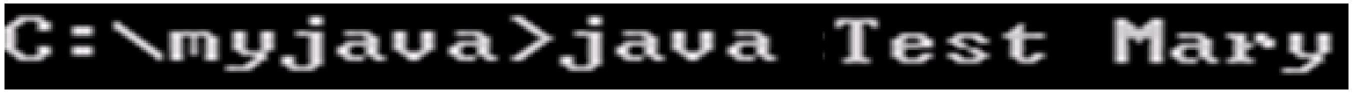
Activity: Constant array

- How will you create a constant array?
- What does it mean when you declare a constant array?

Command Line Arguments

- While executing a java program from the command line, arguments can be sent to the java program from the command line.
- Command line arguments are sent as string arrays through main function’s argument.
- Command line argument counting begins only after the **“java ClassName”.**

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



```
C:\myjava>java Test Mary
```

Activity: Command Line Arguments

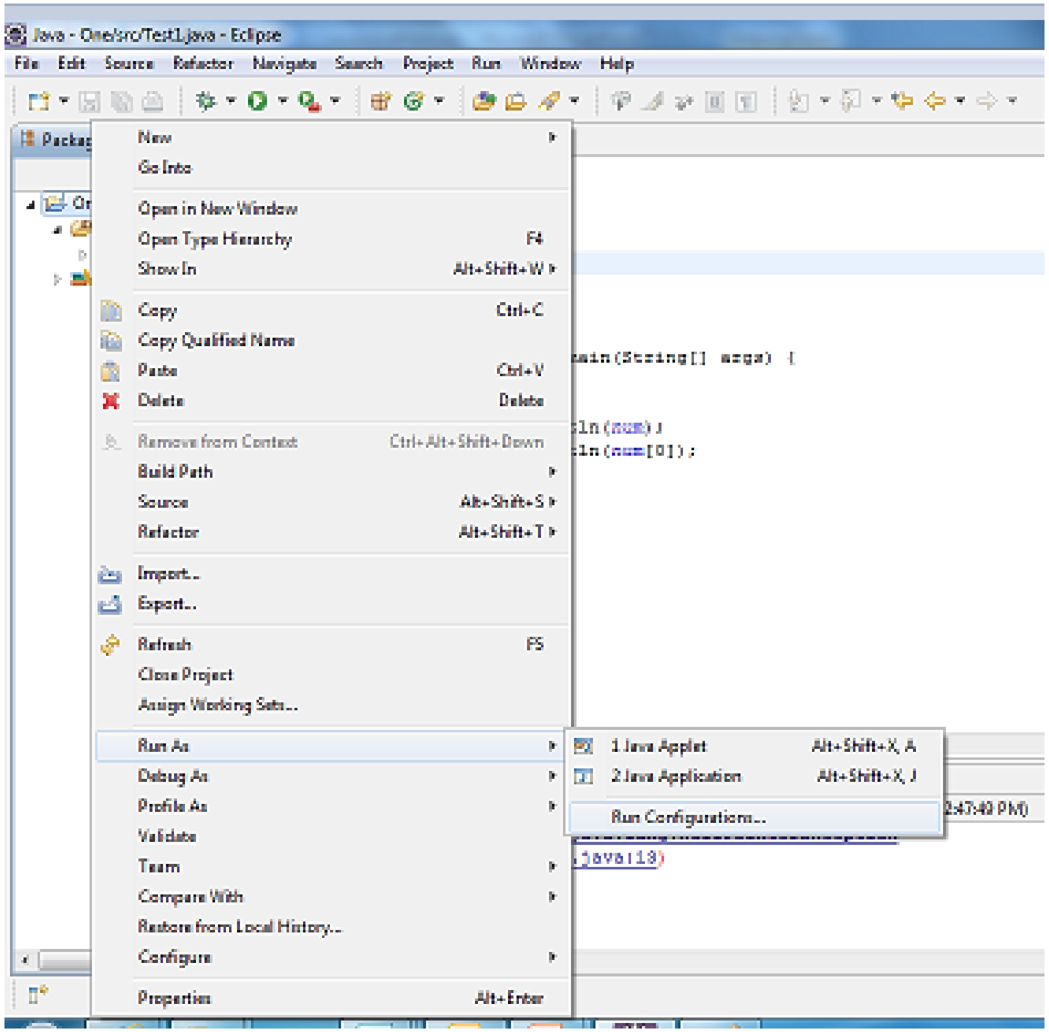
- *Checkout what happens if you run Test with and without supplying command line arguments ?*
- *Why doesn't the first line end up in NullPointerException?*

This is because when there are no arguments passed, the JVM creates a 0 length array and passes to main()→ **`main(new int[0])`**

When an array of length 0 indicates an empty array. Note that this is different from array being null. That is why NullPointerException is not thrown. ArrayIndexOutOfBoundsException is thrown!

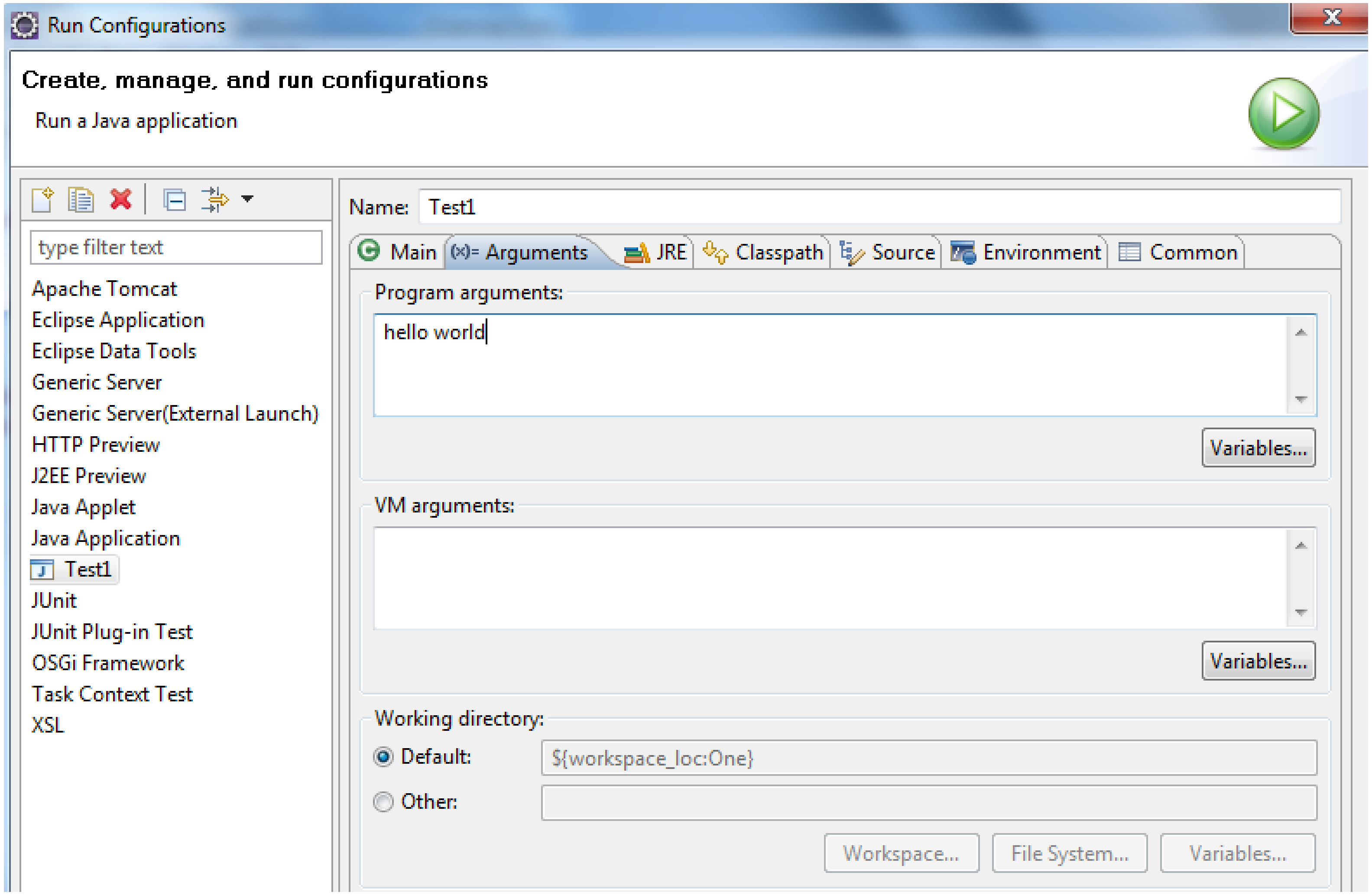
Command Line Arguments in Eclipse

- Right click on the Project and select RunAs→Run Configuration.



```
main(String[] args) {
ln(num);
ln(num[0]);
```

- Move to the 2nd tab called Arguments. In the Program arguments section provide command line arguments separated by spaces



Run Configurations

Create, manage, and run configurations

Run a Java application

Name: Test1

Program arguments:

hello world

VM arguments:

Working directory:

Default: ${workspace_loc:One}

Other:

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);
```

Methods of `String` class:

- `int length()`
- Returns the length of this string

```
String s= new String(“Hello”);
System.out.println(s.length());
//prints 5
```

- `char charAt(int index)`
- Returns the character at the specified index

```
String s="Have a nice day";
System.out.println(s.charAt(0));
// prints H
```

- String concatenation can be done in two ways:
- **+**: Not a method but an operator that can be used with strings (Unlike C++, there no operator overloading in Java. However + is overloaded for strings for programmer convenience.)

```
String s1=“abc”,s2=“def”;
String s3=s1+s2; // returns abcdef
String s4=s1+1; // returns abc1
String s5=s1+”ee”; // returns abcee
String s4=s1+true; // returns abctrue
String s4=s1+’d’; // returns abcd
String s4=null+ s1; // returns nullabc
```

- **`String concat(String str)`**

```
String s1=“java”.concat(“c”);//returns “javac”
```

- To compare if two strings are the same, equals methods are used.
- `boolean equals(Object object)`
- `boolean equalsIgnoreCase(String anotherString)`

We will look at **`Object`** *class later. For our understanding let us replace* **`String`** *for* **`Object`**.

Example:

```
String s1="abc";

String s2="sbc";

String s3="ABC";

s1.equals(s2) ;//returns false

s1.equalsIgnoreCase(s3) );//returns true
```

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 are 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
```

- To work with parts of a string, we have two methods
- Index begins from 0.
- **`public String substring(int beginIndex)`**
- **`public String substring(int beginIndex, int endIndex)`**

Example:

```
“icecream".substring(3); // returns “cream‟
“icecream".substring(0,3); // returns “ice”
```

- To remove with leading and trailing whitespace

  `String trim()`

```
" ice  cream  ‟.trim() // returns ice  cream
```

- Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings
- **`public int compareTo(String anotherString)`**
- **`public int compareToIgnoreCase(String str)`**

Example:

```
String s1="ABC";String s2="acc";
s2.compareTo(s1)
returns 32
s2.compareToIgnoreCase(s1)
returns 1
```

- Converting primitives to **`String`**

```
static String valueOf(XXX b)
```

where **`XXX includes all primitives like byte, short, int, long, float, double, char, boolean.`**

```
 Example:

 int i=String.valueOf(1224);
```

- Tokenizing string

```
String[] split(String regex)

 Example :

 String str="apple,mango,banana";

 String list[]=str.split(",");

 for(String s:list) System.out.println(s);
```

- *Can you guess what these methods do?*
  - **`String toLowerCase()`**
  - **`String toUpperCase()`**

  - **`String replace(char oldChar, char newChar)`**
  - **`String replaceAll(String reg,String replacement)`**

  - **`boolean startsWith(String prefix)`**
  - **`public boolean endsWith(String suffix)`**

Activity: using String class

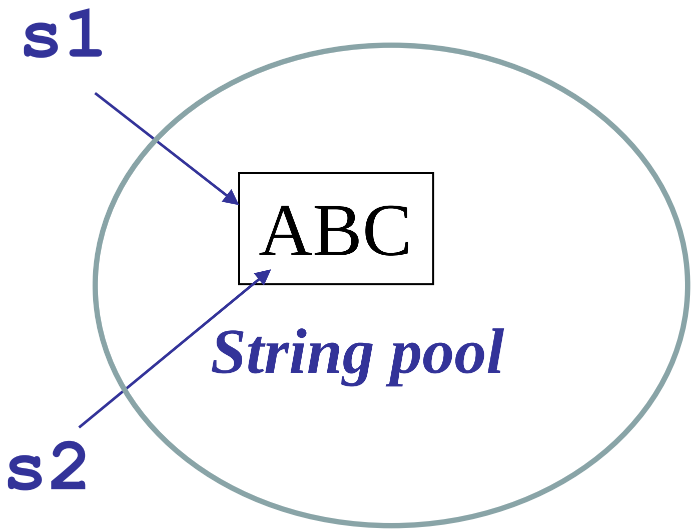
- Can you write code for **`properCase()`** methods now?

Immutability

- Immutability means something that cannot be changed.
- Strings are immutable in Java. What does this mean?
- 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.
- How does this save memory?
- Let us find out….

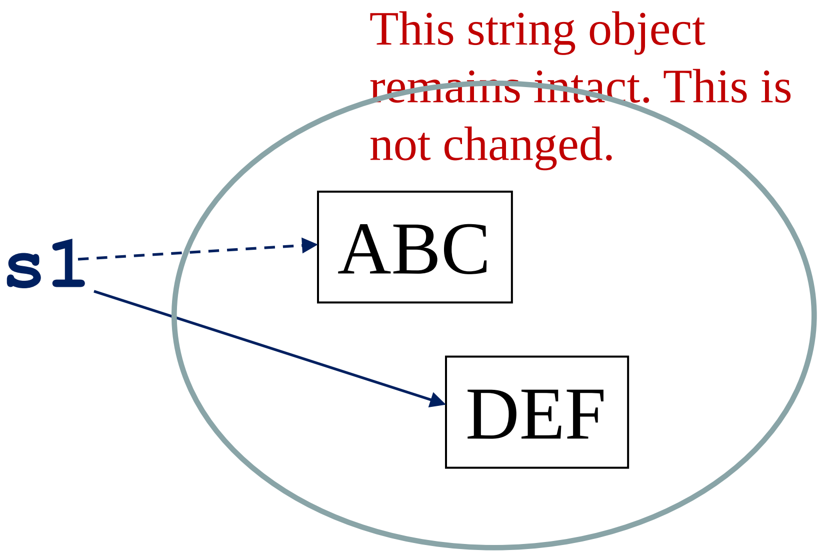
```
String s1="ABC";
String s2="ABC";
```

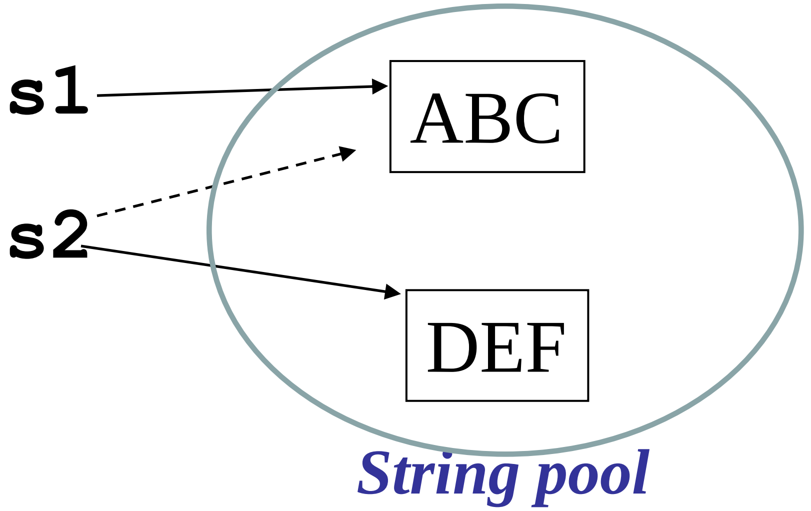
- When Strings are created this way by assigning literals to the variable straight, JVM checks if the string is available in the pool. If not it creates one. Otherwise it assigns it to the existing reference .



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

- When a value of a string reference is changed, a new string is created in the pool and that is assigned to the reference.
- Strings are Immutable Objects.
- That means that, once created, String object cannot be changed!





Assigning string references:

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

Activity: Test your understanding

```
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);
```

Activity: Test your understanding

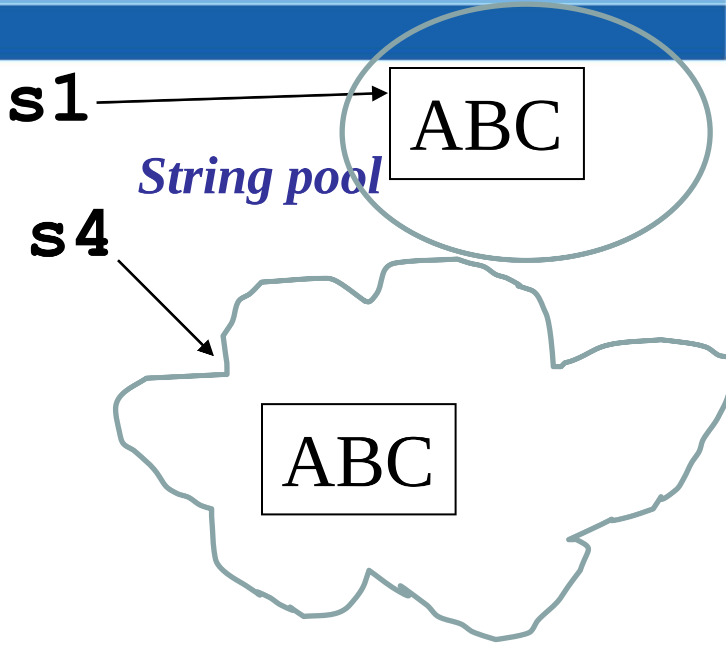
```
String student1= “Mary”;

String studentref=“John”;

String studentref=student1;

student1=null;
```

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

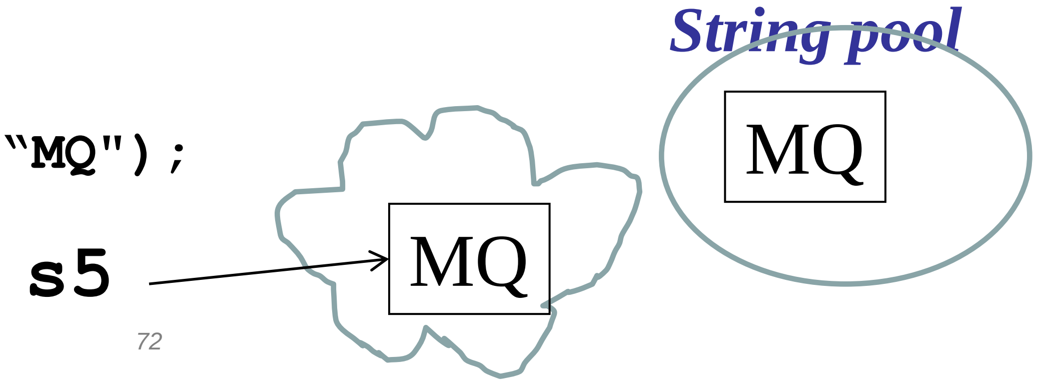
Ans: None

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");`



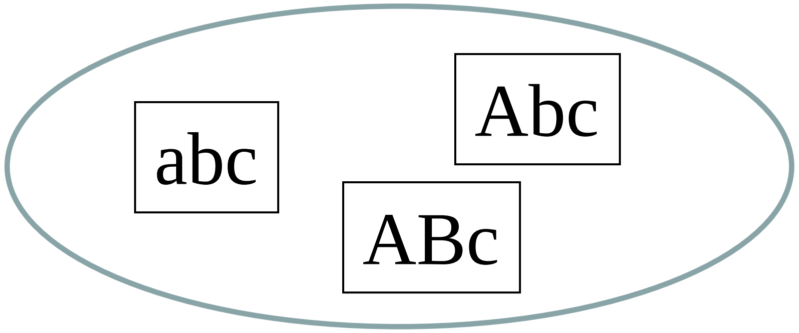
Test your understanding

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

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`** or **`StringBuffer`** class