# Class and Methods-Part2

# Table of Content

- Arguments passing
- Passing primitive type
- Passing reference type
- Passing Arrays
- Var-args
- Accessing var-args
- Formatted Output
- Overloading
- Automatic conversion
- Initializers
- Another place to initialize fields
- Initializing methods to the fields
- Initializations order

# Arguments passing

- Arguments passing in java is always ‘pass by value’.

- Two types of arguments can be passed in a method call
  - Primitive type
  - Reference type

# Passing primitive type

```
public class Test{
public static void swap(int reg1,int
 reg2){
int temp;
temp=reg1;
reg1=reg2;
reg2=temp;    }
public static void main(String args[]){
int r1=10, r2=20;
swap(r1,r2);
System.out.println(“r1=”+ r1);
System.out.println(“r2=”+r2);
}}
```

The program prints r1=10 and r2=20.

# Passing reference type

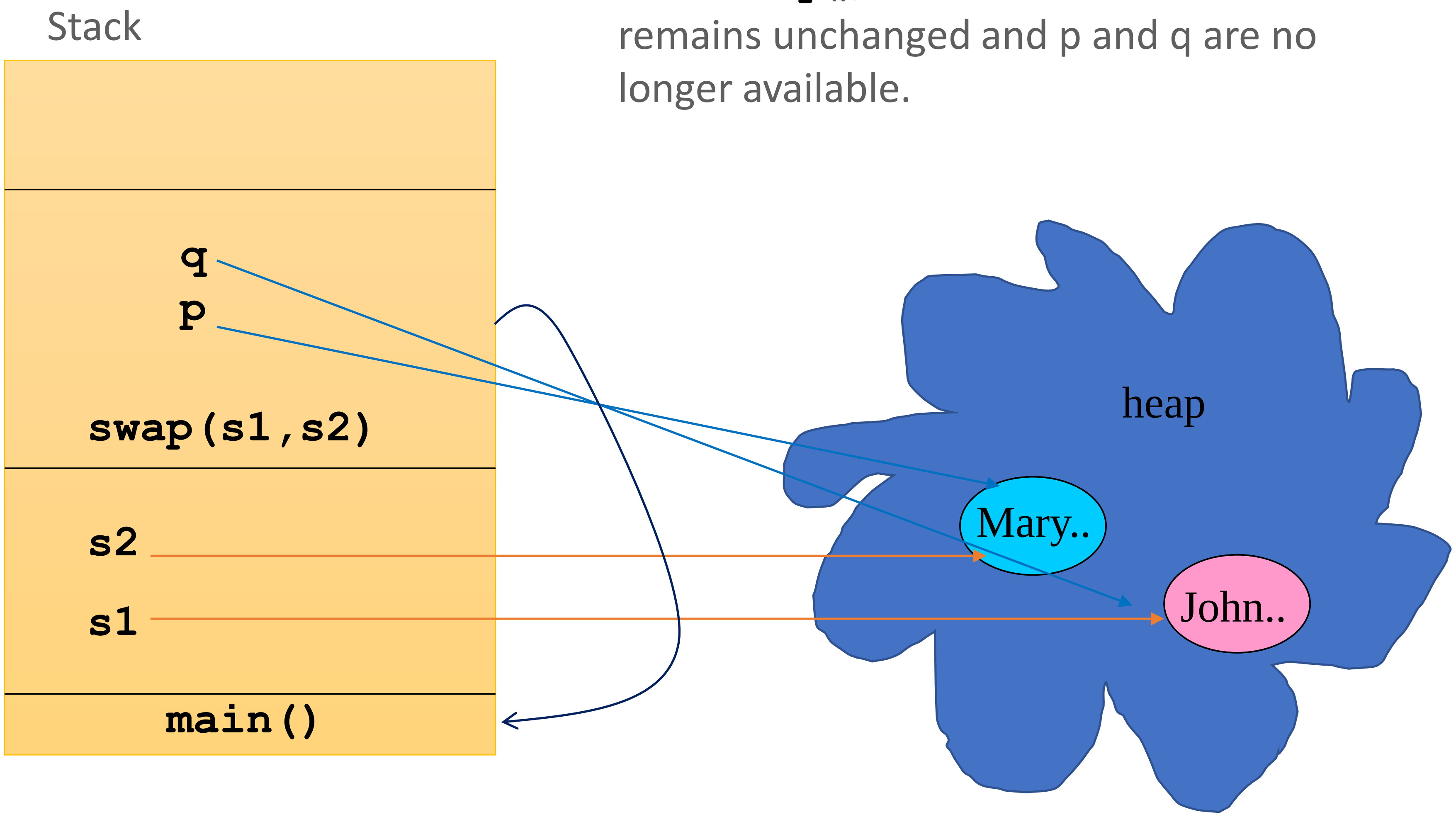
```
public class Test{
public static void swap(Student p,Student q){
Student temp;
temp=p;
p=q;
q=temp;
}

public static void main(String a[]){
Student s1=new Student(“John”);
Student s2=new Student(“Mary”);
swap(s1,s2);
System.out.println(“s1=”+ s1.getName());
System.out.println(“s2=”+ s2.getName()
} }
```

The program prints :

```
s1=John
s2=Mary
```

After **swap**(), s1 and s2 of **main** remains unchanged and p and q are no longer available.



# Test your understanding

*What if we change the value of a member of an object in the method to which we pass a reference?*

```
public class Test{
   public static void change(Student p){
   p.setName(“Mary”);
   }
   public static void main(String args[]){
   Student s1=new Student(“John”);
   change(s1);
   System.out.println(“s1=”+ s1.getName());}
}
```

# Exercise

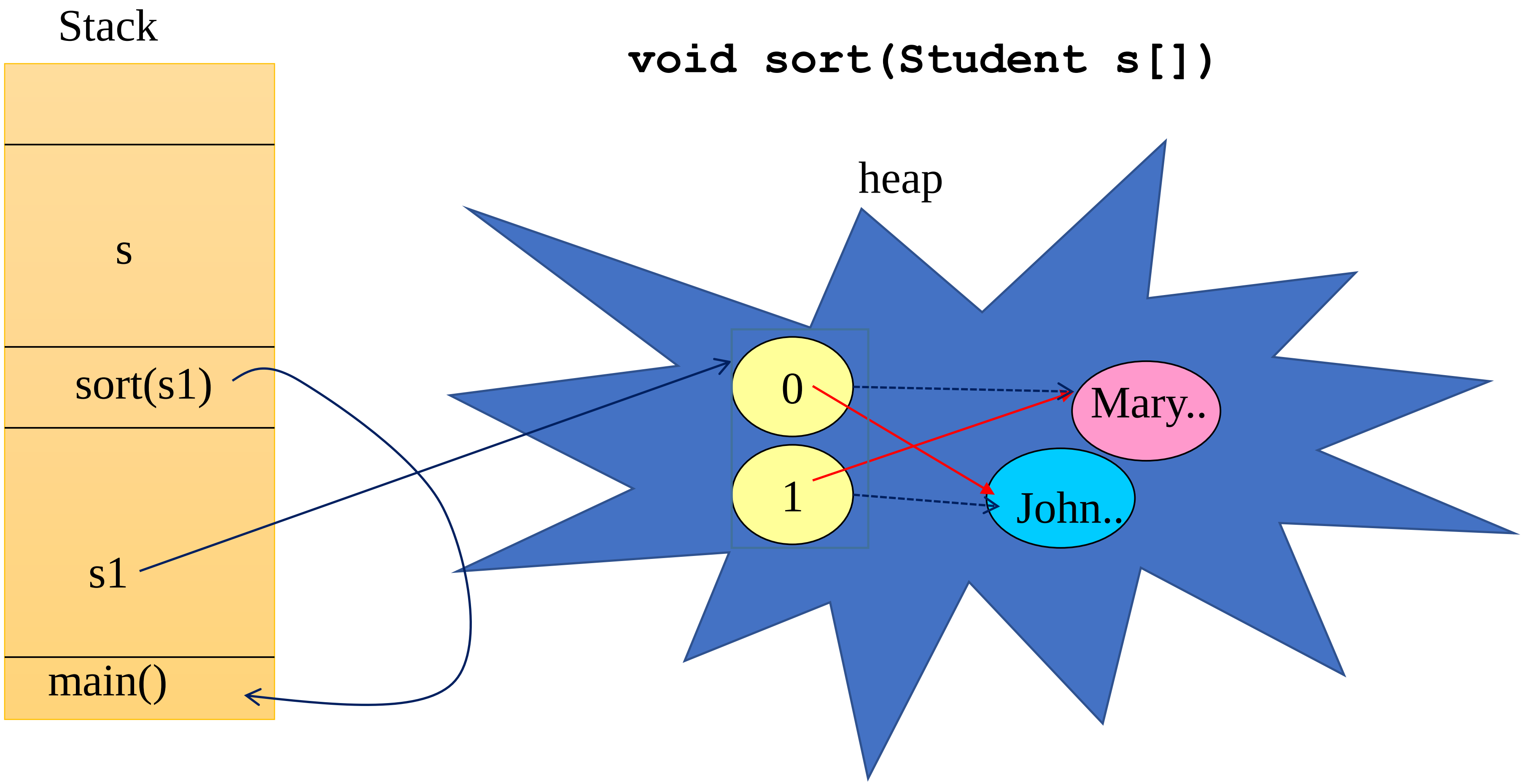
*You have created a Student class in the previous session. Create a student array and populate it with student objects.*

*Have a static method called sort. Pass array of students to the sort method. Sort the array in the method say based on id.*

*(30 mins)*

*Do you find the array in the sorted form when the sort() method call returns back to the main method?*

# Passing Arrays



- Changes made to the array content in the called method are reflected in the calling method.

# Test your understanding

*What is the result when you execute the following?*
*Can you explain what is happening in this case?*

```
class StudentTest{
public static void main(String args[]){
      Student s1[]=new Student[2];
      s1[0]=new Student("Mary");
      s1[1]=new Student("John");
      change(s1);
      for(int i=0;i<s1.length;i++){
      System.out.println("Name:  " +
      s1[i].getName());
      }}
public static void change(Student s[]){
Student temp[]=new Student[1];
temp[0]= new Student("Meena");
s=temp;}  }
```

# Var-args

- Var-args allows a method to take multiple arguments of same type.
- The number of arguments may be 0, one or more.
- In a method only the last argument can be of variable length.
- **void f(int… x)**
  - f()
  - f(1)
  - f(1,2) and so on
- **void go(int c, char… x)**
  - go(1)
  - go(1,’a’)
  - go(1,’b’,’a’,’c’) and so on

- **void list(Student… a)**
  - list()
  - list(new Student(“hari”));
  - list(new Student(“hari”),

    new Student(“rama”));

    and so on

# Accessing var-args

- Compiler interprets var-args like an array.
- Subscript operator is used to access elements in var-args.

```
public class Person{
public Person(String name, String… nicknames)
{
 if(nicknames.length!=0){
 for(String nm:nicknames)
 System.out.println(nm);

 System.out.println(nicknames[0]);
} }}
```

*What are the ways to create Person instance?*

- Another way to write the main method is:

```
public static void main(String... args)
```

# Test your understanding

```
static void vararg1(int[] i)
```

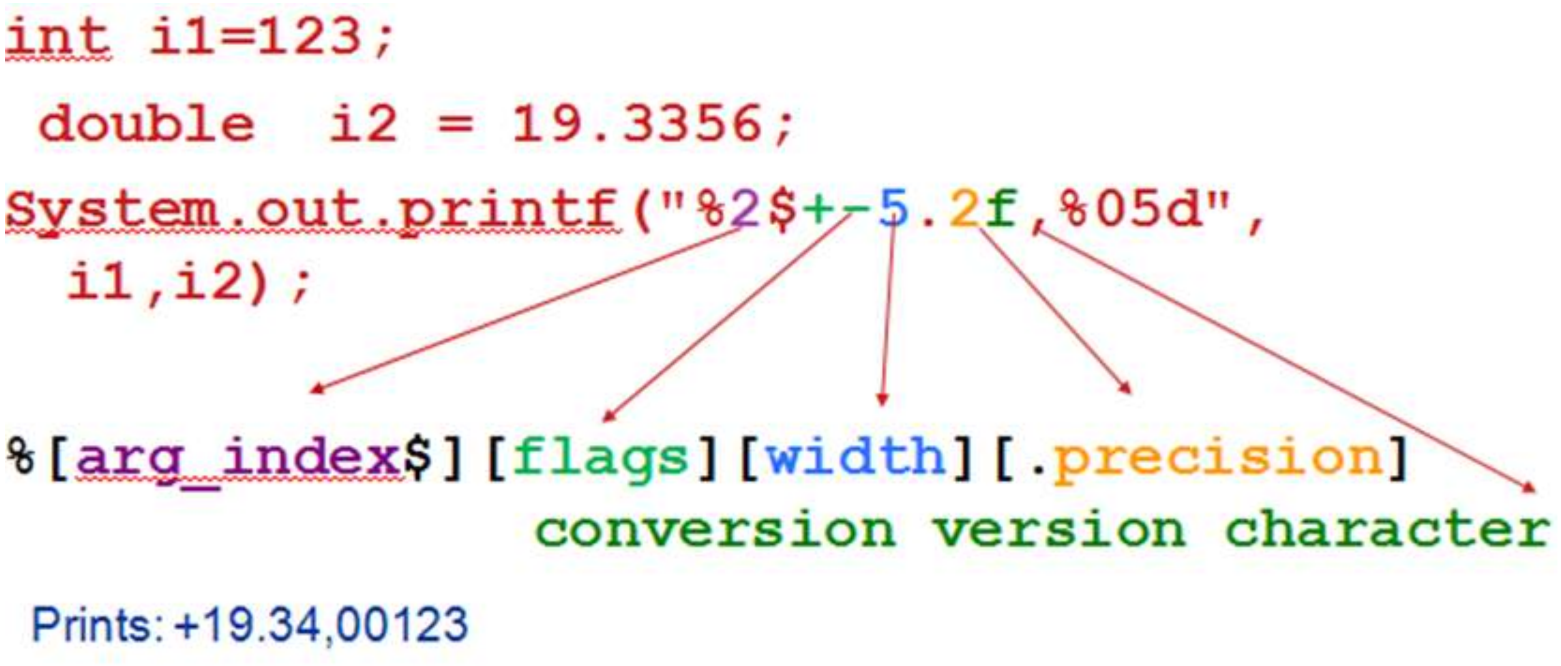
*and*

```
static void vararg2(int… i))
```

*Are both same ?*

# Formatted Output

- Java implements var-args for **`printf`** statement.
- Java also has C-like **`printf`** method that can be used to format output.



```
int i1=123;
 double  i2 = 19.3356;
System.out.printf("%2$+-5.2f,%05d",
  i1,i2);

%[arg_index$][flags][width][.precision]
               conversion version character
```

Prints: +19.34,00123

# Rules

- Available from Java 1.5 onwards.
    - **`argument_index$:`** specifies position of the argument in the argument list, Ex **`1$, 2$`** etc
    - **`flags:`** characters that specify the output format based on the type of output. Ex : **`-`** **`+`** etc
    - **`width:`** positive integer that specifies the minimum number of characters to be written to the output
    - **`precision:`** positive integer usually used to limit the number of characters (after decimal for floating points)
    - **`conversion:`** a formatting character that is specified based on the type argument.

# Conversion characters:

- **'d' (integral):** The result is formatted as a decimal integer
- **'o' (integral ):**The result is formatted as an octal integer
- **'x', 'X' (integral ):**The result is formatted as a hexadecimal integer
- **'e', 'E' (floating point ):**The result is formatted as a decimal number in computerized scientific notation
- **'f' (floating point):**The result is formatted as a decimal number
- **'g', 'G' (floating point ):**The result is formatted using computerized scientific notation or decimal format, depending on the precision and the value after rounding.
- **'b', 'B' (general):**If the argument arg is null, then the result is "false". If arg is a boolean or Boolean, then the result is the string returned by String.valueOf(). Otherwise, the result is "true".
- **'s', 'S' (general ):**If the argument arg is null, then the result is "null". If arg implements Formattable, then arg.formatTo is invoked. Otherwise, the result is obtained by invoking arg.toString().

# Flags

- **-:**  Left justify this argument
- **+:** Include a sign (+ or -) with this argument
- **0:** Pad this argument with zeroes otherwise spaces will be padded to meet the width
- **,:** Use locale-specific grouping separators (i.e., the comma in 123,456)
- **(:** Enclose negative numbers in parentheses

Also conversion character %n can be used for inserting a new line

# Formatted Output Example

```
public class Test{
      public static void main(String[] args) {
        long n = 123456;
        System.out.printf("%d%n", n); //123456
        System.out.printf("%07d %n", n); //0123456
        System.out.printf("%+7d%n", n); //+123456
        System.out.printf("%,7d%n", n); //123,456
        System.out.printf("%+,7d%n", n); //+123,456

        int x=20;
        System.out.printf("%x%n", x); //14
        System.out.printf("%o%n", x); //24

        int y=-20;
        System.out.printf("%+3d%n", y); //-20
        System.out.printf("%(3d%n", y); //(20)

        System.out.printf("%b%n", (x>y)); //true
```

```
    double pi=3.141593;
    System.out.printf("%f%n", pi); //3.141593
    System.out.printf("%.3f%n", pi); //3.142
    System.out.printf("%10.3f%n", pi); //     3.142
    System.out.printf("%010.3f%n", pi); //000003.142
    System.out.printf("%-10.5g%n", pi); // 3.1416

    String s= "Hello";
    System.out.printf("This is %s%n",s);
    // This is Hello
    System.out.printf("%3$s %2$f %1$d %3$s %n",n,pi,s);
    // Hello 3.141593 123456 Hello

    }
}
```

# Exercise

*Write a program that accepts the user's full name , gets the unicode char for each letter and passes these numbers to the method called generatePassword(). The method generatePassword() should reverse each number and then combine all the reversed numbers into one. If the number* has more than *5 digits, then the number must be divided by 5 until* five digit number is reached. *Finally this number is displayed in its octal and hexadecimal notation.*

*Find a method in Java API number that converts the final number that you obtained in the above exercise to hexadecimal number.*

# Overloading

- Overloading refers to the methods in a class having same name but different arguments(types of arguments or number of arguments or order of arguments)
- Signature of a method includes the name of the method and its parameters (excluding the return type).

- Complier resolves the overloaded method using the following steps

  1. **Exact Match**
  2. **Automatic Conversion**
  3. **More Specific Match**
  4. **Ambiguous Match/No Match**

# Exact match

```
public class StudentTest{

public static void display( int regno)
{
System.out.println(“Registration No. "+regno);
}
public static void display(String name)
{
System.out.println(“Name. "+name);
}
public static void main(String str[]){
 Student s1=new Student(“Mary”);
 display(s1.getName());
 display(s1.getRegNo());
}
}
```

# Automatic conversion

*Recall and draw the conversion sequence that we did in “Basic elements of Java” session.*

```
public class StudentTest{
public static void display(long regno){
System.out.println(“Registration No. "+regno);
}
public static void display(String name)
{
System.out.println(“Name. "+name);
}
public static void main(String str[]){
 Student s1=new Student(“Mary”);
 display(s1.getName());
 display(s1.getRegNo());
}}
```

**`int`** automatically convertible to **`long`**

# More specific

```
public class Fee {
int id;
double amtPaid;
void pay(int id,double amt)
{
this.id=id;
amtPaid=amt;
}
void pay(int id,float amt){
this.id=id;
amtPaid=amt; }

public static void main(String[] args) {
Fee f1= new Fee();
f1.pay(123,400);
}
}
```

- `int` automatically convertible to both `float` and `double.`
- But since `float` appears first in the list from `int`, `float` is more specific.

# Ambiguous call

```
public class Fee {
int id;
double amtPaid;
void pay(float id,double amt)
{
this.id=id;
amtPaid=amt;
}
void pay(double id,float amt)
{
this.id=id;
amtPaid=amt; }

public static void main(String[] args) {
Fee f1= new Fee();
f1.pay(123,400);
}
}
```

- **`int`** automatically convertible to both **`float`** and **`double`.**
- In this case the 1st argument 123 is converted to float and 2nd argument 400 to double for the 1st overloaded method. Vice versa happens for the 2nd.
- Hence the call is ambiguous.

# Overloading with var-args

```
class AddVarargs {
static void go(int x, int y)
{
 System.out.println("int,int");
}
static void go(byte... x) {
 System.out.println("byte... ");
}
public static void main(String[] args){
byte b = 5;
go(b,b);
}
}
```

Prints **`int,int`**

- The var-args will be the last argument that will be resolved.

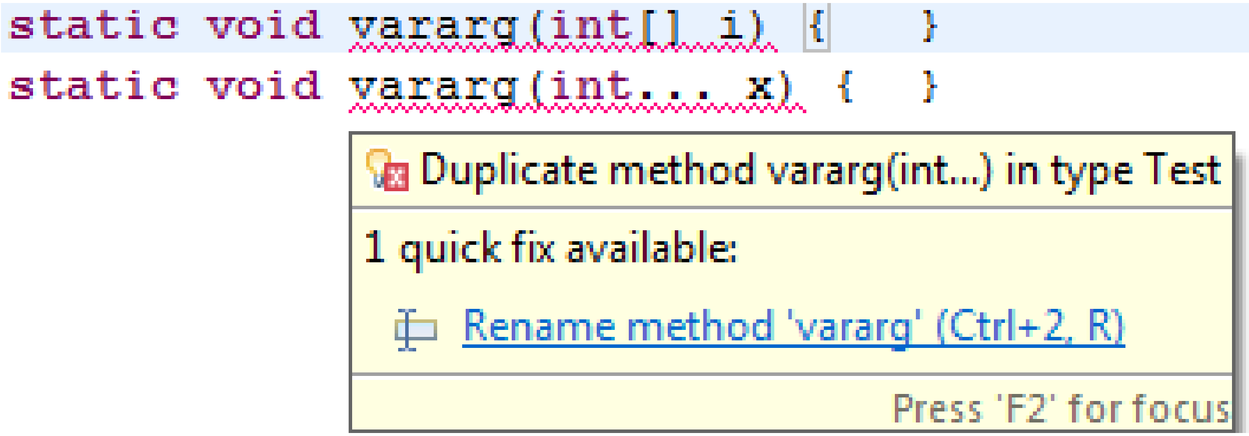
# More on with Var-args

```
static void vararg(int[] i){}
static void vararg(int... x){}
```

cannot be overloaded

Compiler converts:

```
vararg(int... x){}→ vararg(int[] i){}
```



```
static void vararg(int[] i) {    }
static void vararg(int... x) {   }
```

Duplicate method vararg(int...) in type Test

1 quick fix available:

Rename method 'vararg' (Ctrl+2, R)

Press 'F2' for focus

# Activity: Test your understanding

*What will the code print?*

```
public class Test{
static void count(String… objs){
System.out.println(objs.length);
}
public static void main(String[] args) {
   count(“1”, “2”, “3”);
   count(“1”,”2”);
   count(“1”);
   count(null,null,null);
   count(null,null);
   count(null);
}
}
```

# Exercise

*Create a class called Calculator which has different add methods. Overload the methods such that the parameters can be of the following pattern.*

*a)Both are of int data type.*

*b)Both are of double data type.*

*c)First parameter is of int data type and second parameter is of double data type.*

*d)First parameter is of double data type and second parameter is of int data type*

*Create an object to access these methods and invoke these methods with different type of numbers and display the result in the corresponding methods.*

# Initializers

- Initializers are blocks of code used to initialize member variables.
  a) Non-Static Initializers
    - Used to initialize instance variables
    - Invoked every time object is created
    - Syntax
      - **`{ <<statements>>}`**

  b) Static Initializers
    - Used to initialize static variables
    - Invoked once when the class is loaded
    - Syntax
      - **`static { <<statements>>}`**

# Another place to initialize fields

- Another way to initialize variable is by assigning initializing methods  to the fields.
- It is recommended that the **`final`** modifier is added to the initializing method.

*final methods will be discussed in inheritance section*

# Initializing methods to the fields

```
public class Test{
private static int[] numbers=init();
private byte[] bytes=initb();
public final int[] initb(){
bytes= new int[50];
for (int i = 0; i < numbers.length; i++) {
bytes[i] = i;
return bytes;}
static final public int[] init(){
numbers= new int[50];
for (int i = 100; i < numbers.length; i++) {
numbers[i] = i+2;
}
return numbers; }}
```

# Initializations order

```
 class W{
public W(){System.out.println("W constructor");}
}
public class Z{
W w= new W();

{
System.out.println("instance block");
}
static{
System.out.println("static block");
}
public Z(){System.out.println("Z constructor");
}
public static void main(String st[]){
System.out.println("In main");
new Z(); new Z();
}}
```

```
Result :
    static block
    In main
    W constructor
    instance block
    Z constructor
    W constructor
    instance block
    Z constructor
```

*Can you guess what will be printed if you comment both the “new Z()” in main()?*
*How will the output look if you have a static block for W class also?*

*Type the code to find and analyze the results.*
*How many .class files are created?*

# Exercise

*Write a class called AppUser that prompts the user for the following information*

1. *Database URL*
2. *Property file name*
3. *User name*
4. *Password*

*Since these are Database URL and Property file name are common for all users and hence they have to be initialized only once when the code is executed.*

*User name and password are specific to each AppUser object.*

*Test the application by creating 2 users and print all the details entered by the user.*
*(15 mins)*

# Summary

- Arguments passing in java is always ‘pass by value’.
- Two types of arguments can be passed in a method call - Primitive type and Reference type.
- Var-args allows a method to take multiple arguments of same type.
- Compiler interprets var-args like an array.
- Java has printf method that can be used to format output.
- Overloading refers to the methods in a class having same name but different arguments.
- Signature of a method includes the name of the method and its parameters.
- Initializers are blocks of code used to initialize member variables.
- Non-Static Initializers and Static Initializers are the two types.