Class and Methods-Part2

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

- In java, the arguments are passed by values -- whether we pass primitive type or reference type.
- Let us understand this with examples.

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.

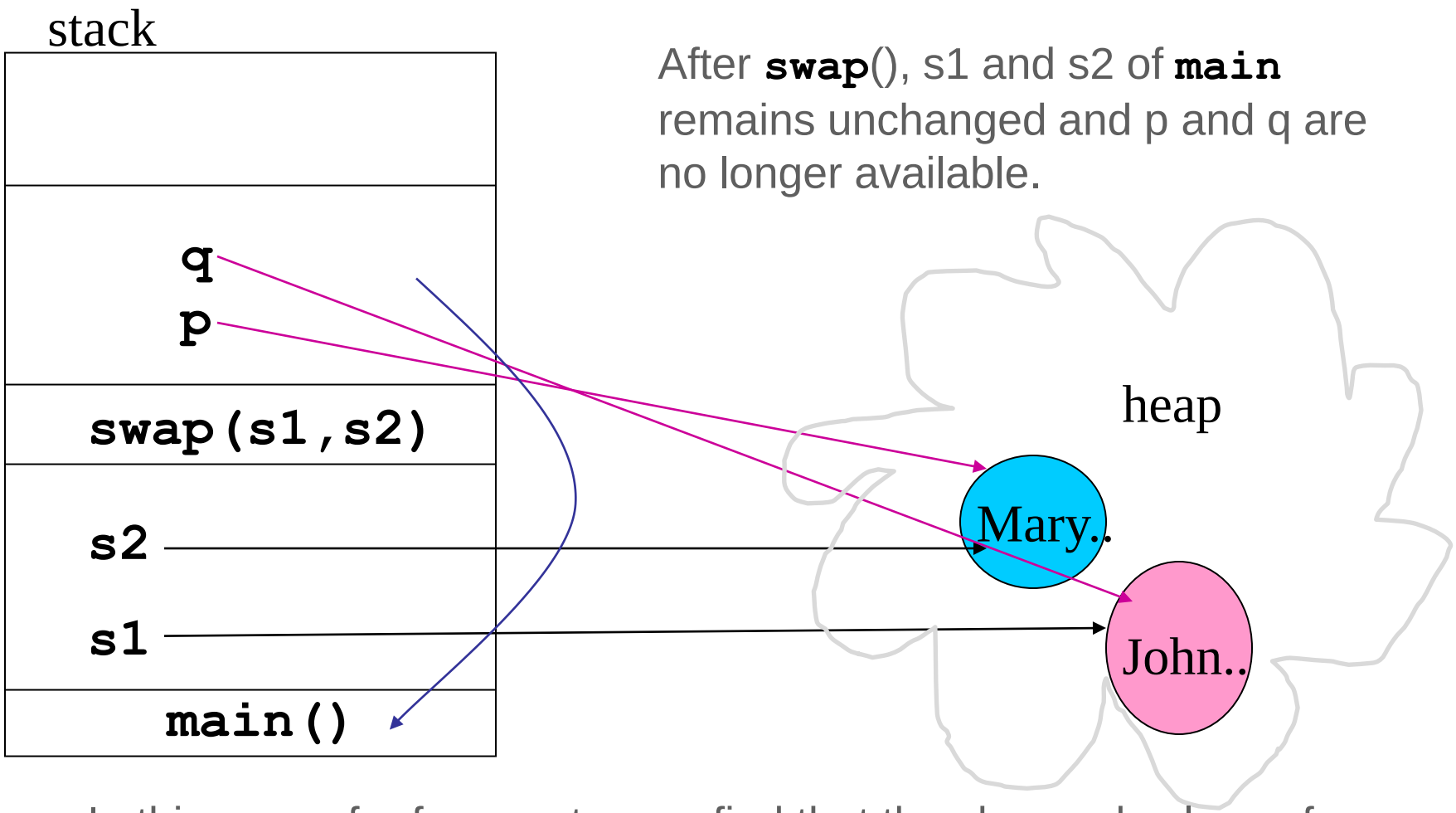
- In the example in the previous slide, two arguments of **`int`** type are passed to a **`swap`** method from **`main`**.
- In **`swap`** method, the value of the 2 arguments passed are exchanged.
- When the method returns to **`main`**, the values changed are not reflected.
- In this case of primitive, changed values of arguments passed are not reflected back in the calling method.
- Hence primitives are passed by value.

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



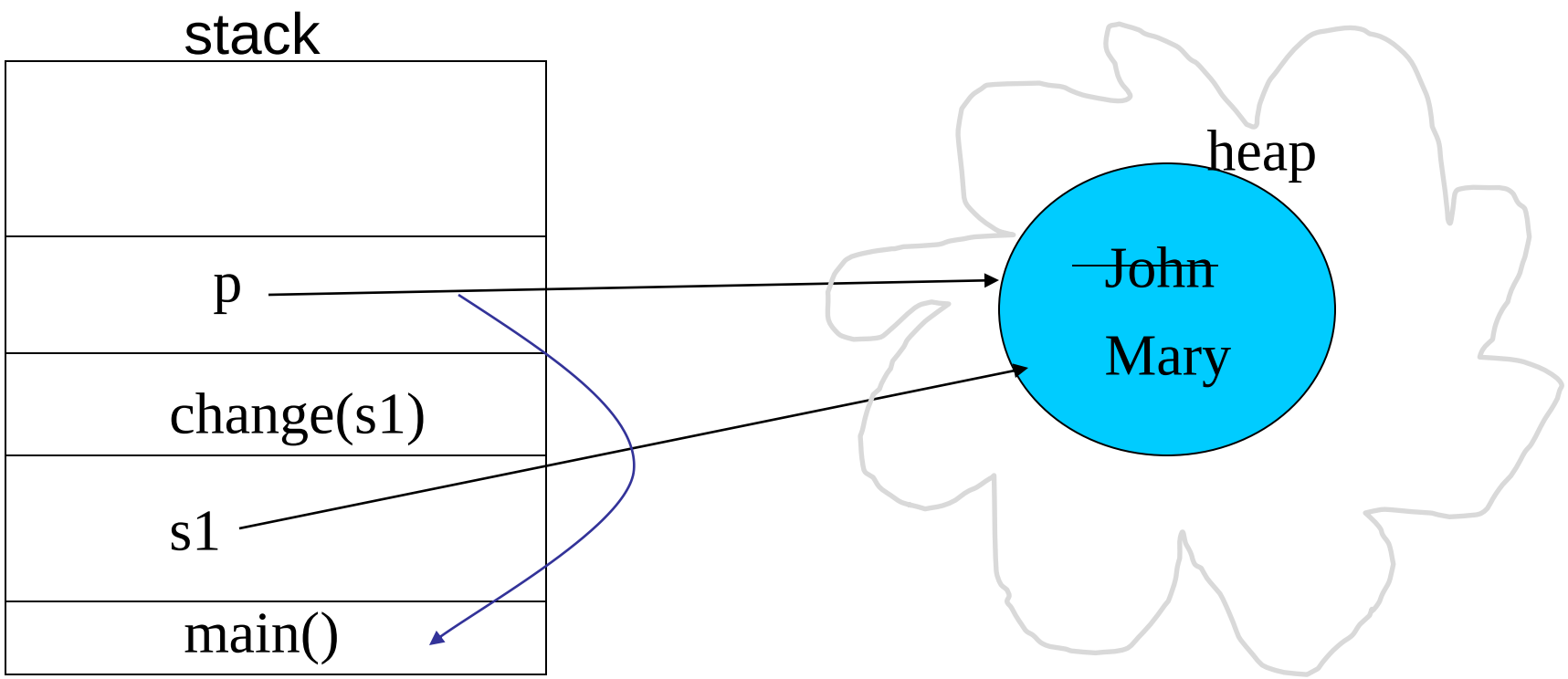
- In this case of reference too we find that the changed values of arguments passed are not reflected back in the calling method.
- Hence references are also passed by value

Changing the member variable value of an object reference that is passed

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());}
}
```

Prints Mary.

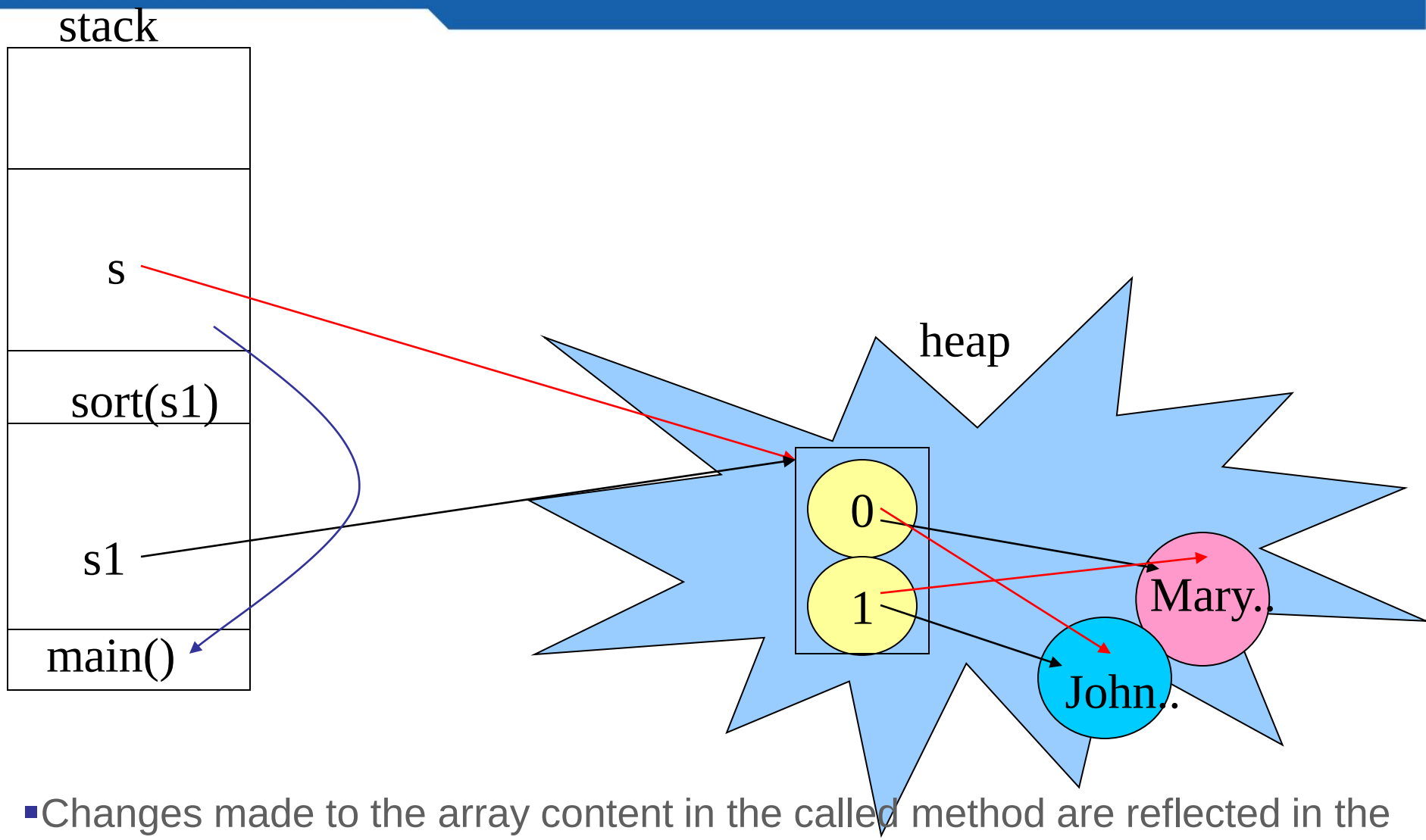


- Since s1 in main and p in change points to the same student object in the heap, when name attribute is changed, the change happens in the object which is in the heap.
- Change made to the member variable of a object in the called method is reflected in the calling method.

Activity : What about arrays?

- Suppose we were to write a sort method that takes an array of Student objects and sorts the array based on their names.
- Will the change be reflected in the calling 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 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. Therefore accessing the var-args is like accessing array elements. Either for loop or enhanced for-loop could be used to iterate through var-args.
- 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.
- Format specifiers :

  values within **`[]`** *are optional*

  `%[argument_index$][flags][width][.precision]conversion`
  - **`argument_index$:`** 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

Extract from Java documentation

`'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 *any type*	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().

Only a few covered at this point.

Flags

- `-`: Left justify this argument
- `+`: Include a sign (+ or -) with this argument
- `0`: Pad this argument with zeroes
- `,`: Use locale-specific grouping separators (i.e., the comma `i`n 123,456)
- `(`: Enclose negative numbers in parentheses

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

Example 1

```
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

Example 2

```
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("%10.3f%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

    }
}
```

Overloading

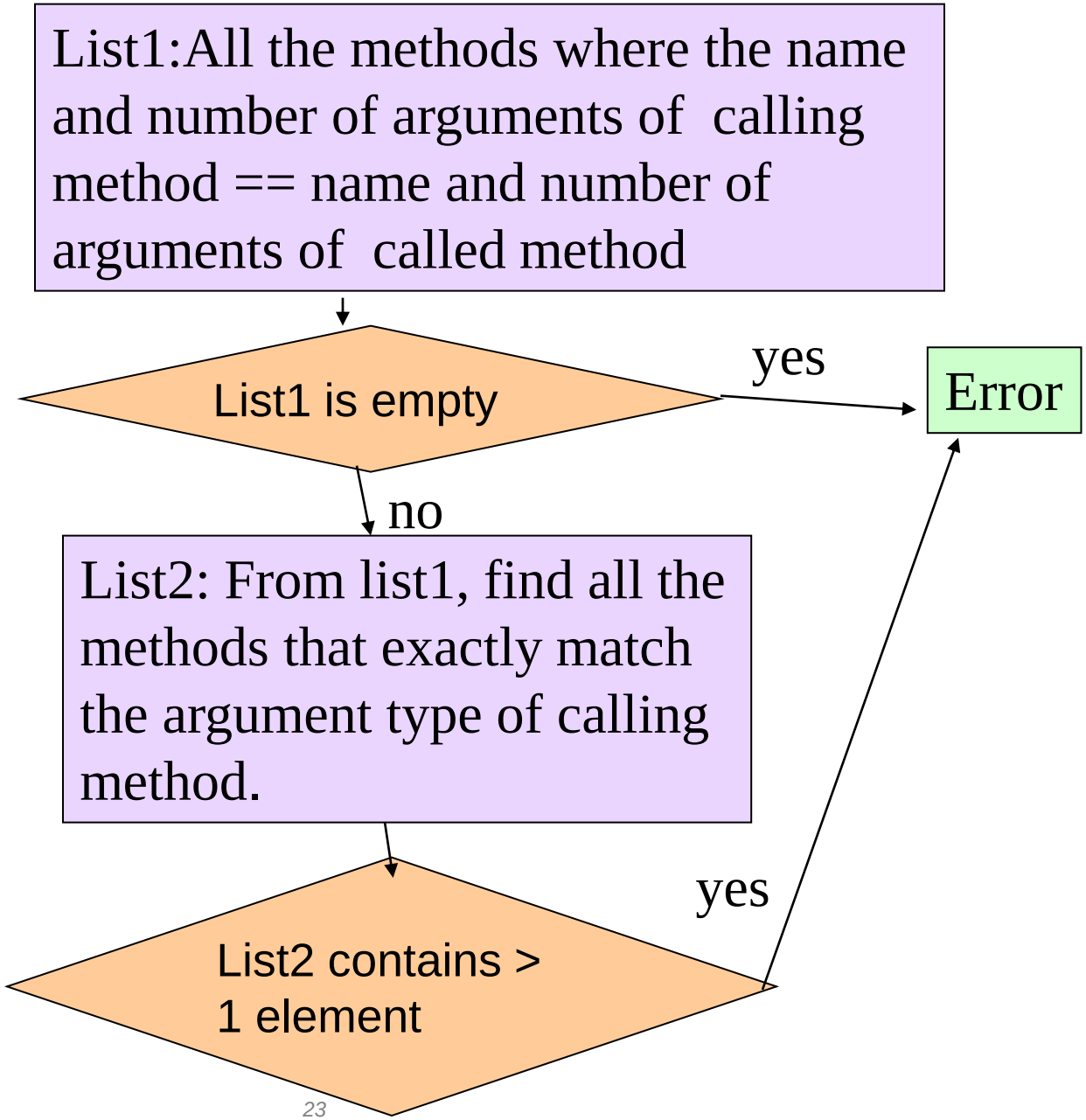
- Like many OOPs languages, Java also provides method overloading feature.
- Overloading refers to the methods in a class having same name but different arguments.
- If same method names are used to define multiple methods in a class, then the methods are said to be overloaded.
- Overloaded methods must have same name but there must be a difference in other parts of the signature.
- Signature of a method includes the name of the method and its parameters (excluding the return type).

Overloading Rules and Resolution

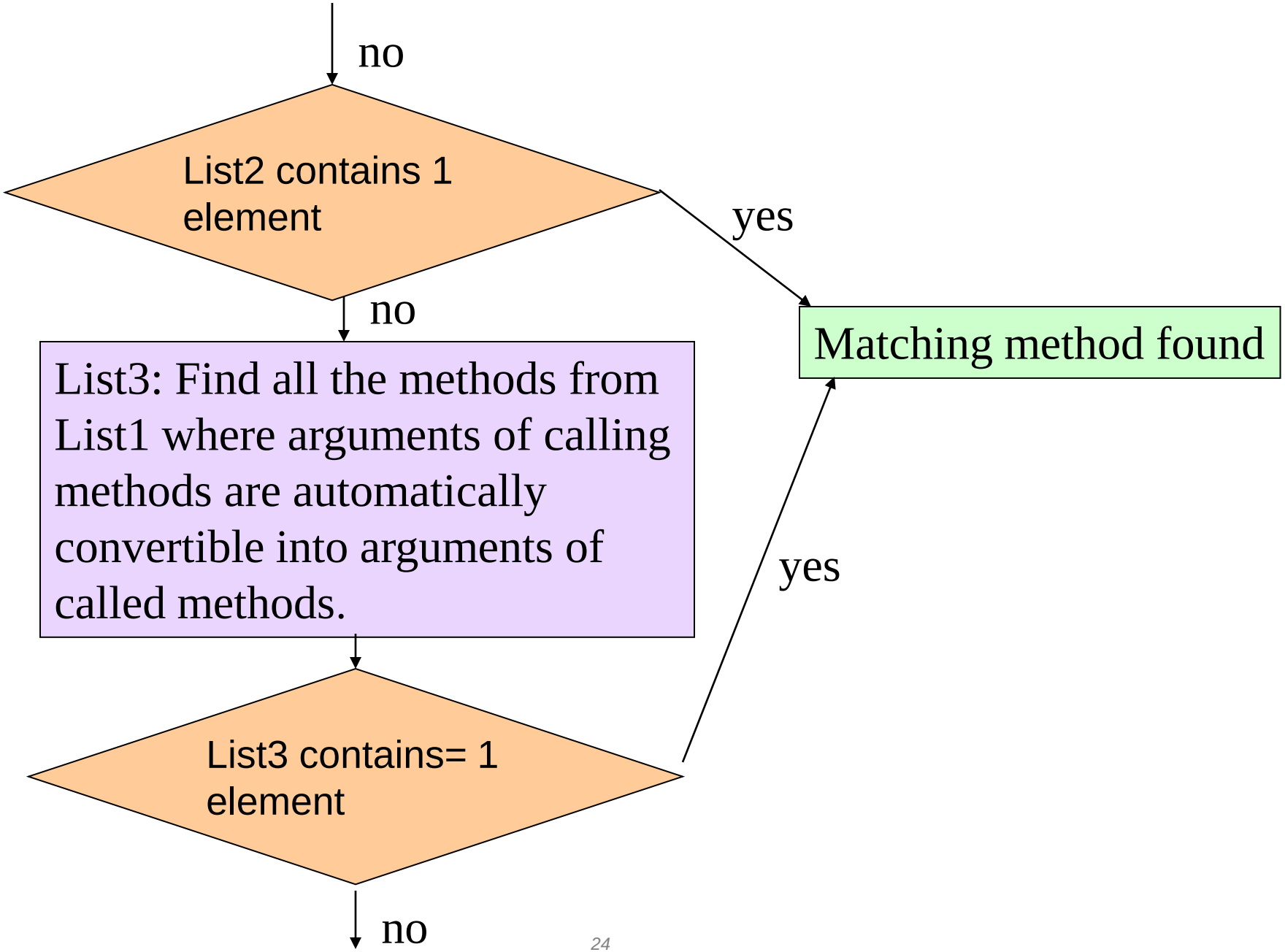
- Same method names in a class having different
  - types of arguments or
  - number of arguments or
  - order of arguments

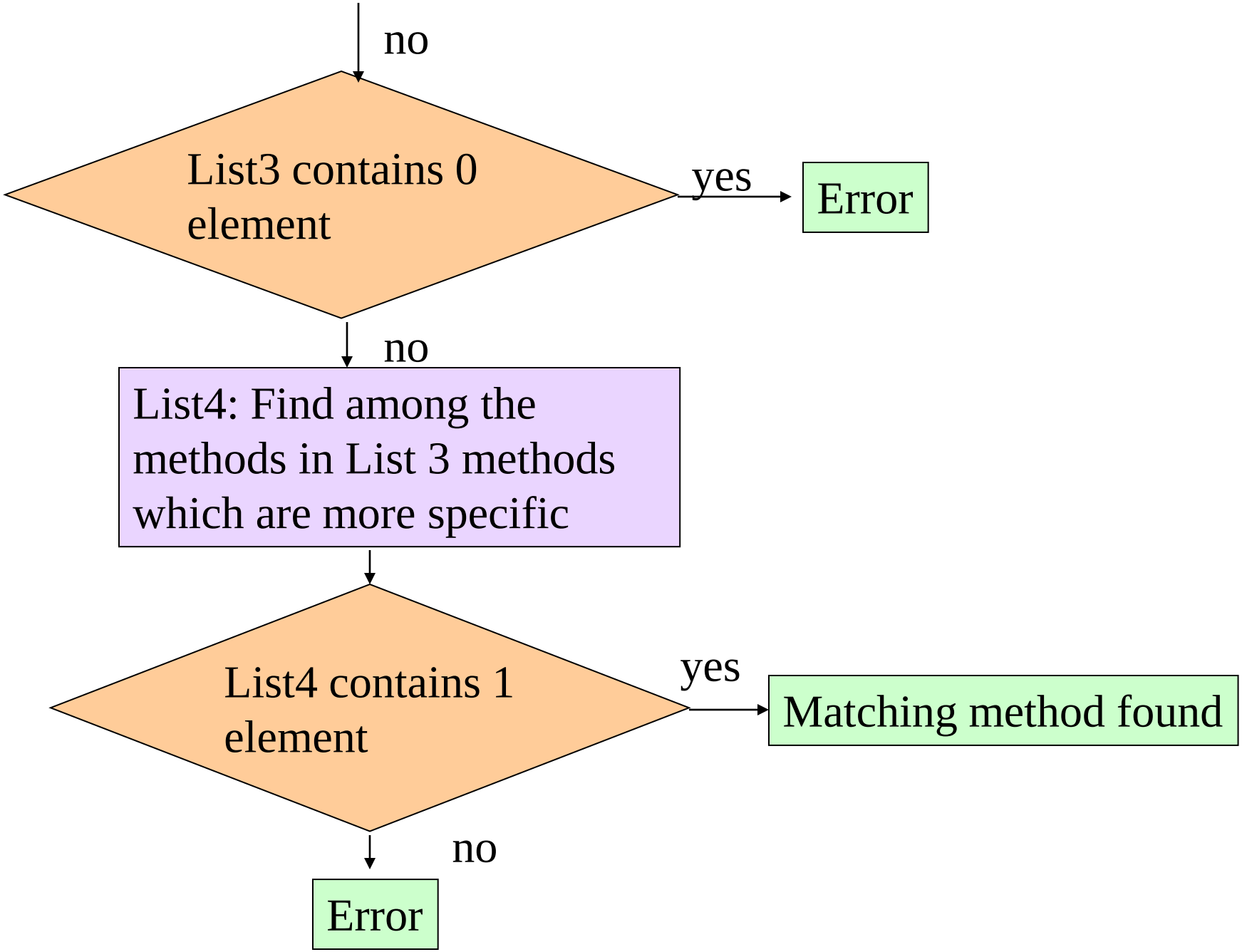
- Complier resolves the overloaded method using the following steps
  1. Exact Match
  2. Automatic Conversion
  3. More Specific Match
  4. Ambiguous Match/No Match

Resolution Flow



Error: No matching method found


no
List2 contains 1 element
yes
no
Matching method found
List3: Find all the methods from List1 where arguments of calling methods are automatically convertible into arguments of called methods.
yes
List3 contains= 1 element
no


no
List3 contains 0 element
yes
Error
no
List4: Find among the methods in List 3 methods which are more specific
List4 contains 1 element
yes
Matching method found
no
Error

Example

- Instead of thinking of names for each sort method, we just have one sort and distinguish the sorts based on the argument we send.

```
public class CollegeUtility {
public static void sort(Student s[]){..}
public static void sort(Faculty c[]){..}
public static void sort(Facility c[]){..}
}
```

Overloading sort method – much simpler

Example for 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());
}
}
```

Example for automatic conversion

Do you recall 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());
}}
```

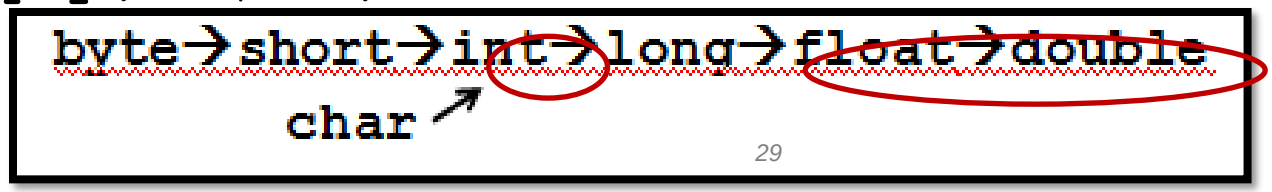
`int` automatically convertible to `long`

Example for 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 is both **`float`** and **`double`**.
- But since **`float`** appears first in the list from **`int`**, **`float`** is more specific.



Example for 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 is 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.
- Reason for this is simply because var-args was introduced only from java 1.5.

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

Tell me how?

- If compiler does this conversion, how does the call **`vararg(1,2,3)`** work then? **`vararg(1,2,3)`** will not match the if the argument is of type array!

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

- Compiler also converts the call statement to make it pass an array instead of commas separated ints, as shown below

  `vararg(1,2,3}`→ `vararg(new int[]{1,2,3}`

Test your understanding

What will the code print?

```
public class Test{
static void count(int…  objs)
{
System.out.println(objs.length);
}
public static void main(String[] args) {
Fee f1= new Fee();
f1.pay(123,400);
count(1, 2, 3);
count(1,2);
count(1);
}
}
```

Prints:
3
2
1

```
public class Test{
static void count(String… objs)
{
System.out.println(objs.length);
}
public static void main(String[] args)
 {
Fee f1= new Fee();
f1.pay(123,400);
count("1", "2", "3");
count("1","2");
count();
}
}
```

Prints:
3
2
0

Why?

Tell me why?

The same code behaves differently for references and primitive type var-args. How is it different for references?

The call to count in the previous slide gets converted as follows:

- **`count(null,null,null)`**
- **`count(null,null);`**
- **`count(null);`**

The last call treats **`null`** to be set to the array object rather than consider one of the elements of array as **`null`**. Therefore, the invoking **`length`** on **`null`** object throws **`NullPointerException.`**

For the code print 1 the call must be

- **`count(new String[] { null });`** → prints 1

Some more on var-args

```
public class Test{
public static void f(double... d){
System.out.println("doubles");
}
public static void main(String[] args)  {
f(1.2, 2.2); // doubles
f(1,2); //doubles
}
}
```

- Compiler converts var-args call depending on the type of the var-args function defined.
- In the above case, call to **`f(1,2)`** is converted to

```
f(new int[]{1, 2})
```

Tell me why?

```
public class Test{
public static void f(double... d){
System.out.println("doubles");
}
public static void f(int... i){
 System.out.println("ints");
}
public static void main(String[] args)  {
f(1.2, 2.2); // doubles
```

~~f(1,2);~~ → The method f(double[]) is ambiguous for the type Test

```
}}
```

This is because compiler does not know if it should **`convert f(1,2)`** into **`int`** array or **`double`** array

Array of varargs

- The example demonstrates how to declare array of var-args.
- The display method takes array of var-args.

```
public class Test {
void display(String[]... vals) {
 for(int i = 0; i < vals.length; i++) {
  for(int j = 0; j < vals[i].length; j++) {
  System.out.println(vals[i][j]);
 }  }
}
public static void main(String[] args) {
Test t = new Test();
String[] s1 =new String[]{"Red,","Yellow"};
String[] s2 = new String[]{"Mars","Jupiter"};

t.display(s1, s2);
}
}
```

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>>}`

Why are they needed?

- The declarations and initialization of fields can be done in same line like this: `int var=1;`
- Initializers are required for initializations that require a set of java statements for computing the initialization variable.
- For instance if the initial value is to be read from a file, then the set of file statements can be put in the initialization block. Another place where this may be required in the use of for-loop for initializing arrays.
- The Compiler copies instance initializer block into every constructor. Therefore, this can be used to share a block of code between multiple constructors.
- The static initializer is the only place to initialize static fields in cases where initialization exceeds more than a statement.

Example

- This is an example that creates fee list for the student from rollnumber 1 to 50, feeList[0] represents marks for student 1 and so on.
- feeList[0] is initialized to 100, which is the minimum that a student has to pay.
- Constants have to be initialized in the place of their declaration. If there is a static block their initialization can be done in this block too.

```
public class FeeList{
public static final int MAX;
private static int feeList[]= new int[MAX];
private static int count=-1;
public String name=null;
static{
MAX=50;
for(int i=0;i<MAX;i++)
feeList[i]=100;
}
```

```
{
if(count>=MAX){
{System.out.print("cannot exceed 50");
System.exit(0);}
count++;
}
}

Test(){}
Test(String nm){
this.name=nm;
}
Test(String nm, int mar){
this.name=nm;
markList[count]=mar;
}
}
```

Max of 50 students. Bounds checking done here

Another place to initialize fields

Can you list out all the places where we can initialize instance variables?

- 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

Example

```
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

- It is important to understand when each of the initialization takes place.
- Static blocks are executed only once when the class is loaded while instance blocks are invoked each time the instance is created.
- First static initializations happen in the order of the declaration.
- Then instance initializations happen. And the appropriate constructors are invoked. (This is obvious since compiler copies instance block into the constructor)
- Static initializations does not require instance to be created. It just requires invocation of any method of the class (static).

Understand by example

Can you guess what the output will be?

```
public 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

```
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 'new Z()' in main()?
How will the output look if you have a static block for W class also?

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