PYEBOOK

STUDY GUIDE

AP® CSA

Table of Contents

Table of Contents

Table of Contents

1. System.out.print Basics

PRINTING

CONCEPT

1. `System.out.print(<message>);`

Prints the message & the cursor stays on the same line

2. `System.out.println(<message>);`

Prints the message & the cursor moves to the next line

EXAMPLE

Consider the following code snippet:

```
System.out.println("PyeBook says,");
System.out.print("Hi ");
System.out.println("Student");
System.out.print("All the best!");
```

Output:

```
PyeBook says,
Hi Student
All the best!
```

OPERATORS

CONCEPT

`+ operator` is used for concatenation in the print statement.

NOTE

The expression inside the print statement is evaluated from **based on the operator precedence**.

EXAMPLE

Consider the following code snippet:

```
int val = 4;
System.out.println("val is "+val);
System.out.println("val+2 "+val+2);
```

Output:

```
val is 4
val+2 42
```

Consider the statement:

`System.out.println(val+3+" is not "+val+3);`

Expression	Evaluated as
`System.out.println(val+3+" is not "+val+3);`	`7`
`System.out.println(7+" is not "+val+3);`	`“7 is not "`
`System.out.println("7 is not "+val+3);`	`“7 is not 4”`
`System.out.println("7 is not 4”+3);`	`“7 is not 43”`

The final output is:

`7 is not 43`

2. Escape Sequences

CONCEPT

1. `\n`

Moves the cursor to a new line

2. `\\`

*Prints *

3. `\"`

Prints "

EXAMPLE

Consider the following code snippet:

```
System.out.println("S1\n0k");
System.out.println("S2\\0k\\");
System.out.println("S3\"0k\"");
System.out.println("S4\n\"0k\"");
```

Output:

```
S1
0k
S2\0k\
S3"0k"
S4
"0k"
```

3. Basic Data Types

CONCEPT & EXAMPLE

Data type	Used for	Size	Example
int	Storing integers	4 Bytes	`-2, 0, 13`
double	Storing decimals	8 Bytes	`-0.29, 1.6, 19.0`
boolean	Storing true or fasle	1 Bit*	`true` or `false`

4. Variable Creation

CONCEPT & EXAMPLE

General naming scheme:

1. No special character except for <underscore>
2. Can contain digits but cannot begin with one
3. Cannot be Java reserved keywords

Type	Practice	Example
Variable	Lower camel case	`int varName1`
Method		`double avgScore`
Class	Upper camel case	`class JavaProg`

General Practices for naming

NOTE

Converting from a broader data type (higher size) to a narrower data type (lower size) may result in loss of information and therefore is not allowed.

Hence, **`int value = 2.5;`** will give a type mismatch error since 2.5 is considered as a double.

But, **`double value = 10;`** will be allowed since 10 is considered as an int and can be stored in a variable that occupies more space.

5. Arithmetic Operator

1. + (Addition)
2. – (Subtraction)
3. * (Multiplication)
4. / (Division)
5. % (Remainder, aka Modulus)

DIVISION OPERATOR (÷)

CONCEPT

NOTE

The division operator gives results based on the data type of operands in the operation.

Type 1 (A)	Type 2 (B)	(A/B)	Example
int	int	**int**	`5/2 = 2`
double	int	**double**	`(5.0)/2 = 2.5`
int	double	**double**	`5/(2.0) = 2.5`
double	double	**double**	`(5.0)/(2.0) = 2.5`

Table of all possible operand's data type combinations for the division operation

EXAMPLE

Expression	Result (val)
`int val = 9/3`	`3`
`int val = 10/4`	`2 (not 3)`
`int val = 9/7`	`1 (not 1.28)`
`double val = 10/4`	`2.0 (not 2.5)` `int/int => int` Common Mistake
`double val = (10.0)/4`	`2.5`
`int val = (10.0)/4`	`Error!` Can't assign double to an int

MODULUS OPERATOR (%)

CONCEPT

The modulus gives the remainder when one number is divided by the other.

EXAMPLE

Expression	Result
`10%4`	`2`
`8%15`	`8`
`24%6`	`0`

DID YOU KNOW!?

The modulus and the division operator have some nice results when used alongside the number 10.

```
<value> % 10 = <last-digit>
<value> / 10 = <number w/o last-digit>
```

This could be very helpful when trying to extract digits of any given number.

Number	Mod 10	Div 10
129	9	12
12	2	1
1	1	0

6. Type Casting

CONCEPT

Explicitly (or implicitly) telling Java to consider a variable, a constant or an expression as another data type.

```
<data-type> <var> = (<data-type>) <value>
```

NOTE

It is important to put the data type to which you want to type cast in ( ) before the value.

EXAMPLE

Expression	Result (val)
`int val = (int)5.6`	`5 (not 6)`
`double val = (double)10/4`	`2.5`
`double val = (double)(10/4)`	`2.0* (not 2.5)`

*Here you are type casting the answer of `10/4`. Which means `10 over 4` will be evaluated first as `2` (since both are integers) and then `2` will be type casted to a double which will be `2.0`

7. Operator Precedence

CONCEPT

This section defines the priority which need to be given to the operators in an expression.

Priority*	Operators
1	`(` or `)`
2	`*` or `/` or `%`
3	`+` or `-`

Lower the priority higher the precedence

NOTE
In case the operators have same priority, we proceed with the evaluation from left to right.

EXAMPLE

Steps for evaluation
1. Selecting the expression which has operators with highest priority.
2. Resolve it (one operator at a time).
3. Repeat 1 & 2 until we've the answer.

Consider the following expression:
`9 + 5 / 4 * 2 % 6 - 2`

Expression	Evaluated
`9 + 5 / 4 * 2 % 6 - 2`	`5 / 4 = 1`
`9 + 1 * 2 % 6 - 2`	`1 * 2 = 2`
`9 + 2 % 6 - 2`	`2 % 6 = 2`
`9 + 2 - 2`	`9 + 2 = 11`
`11 - 2`	`11 - 2 = 9`

The expression `9 + 5 / 4 * 2 % 6 - 2` evaluates to **9**.

8. Assignment Operator

SIMPLE ASSIGNMENT

CONCEPT

This is also known as right to left assignment, wherein the contents on the right are assigned to the variable on the left.

```
<var> = <value> or <expression>
```
General expression for simple assignment

NOTE
The LHS can only be a single variable.

EXAMPLE

Expression	Valid or Invalid
`a = b`	Valid
`a * b = c`	Invalid
`a = b * c + 1`	Valid

COMPOUND ASSIGNMENT

CONCEPT & EXAMPLE

Expression	Interpreted as
`a += b`	`a = a + b`
`a *= b`	`a = a * b`
`a -= b`	`a = a - b`

NOTE
The difference in compound assignment when compared to simple assignment is that it automatically performs type casting.

9. Unary Operators

CONCEPT

Type	Written as	Interpretation

Pre-increment	`b = ++a`	`S1: a = a+1`
		`S2: b = a`

Post-increment	`b = a++`	`S1: b = a`
		`S2: a = a+1`

Pre-decrement	`b = --a`	`S1: a = a-1`
		`S2: b = a`

Post-decrement	`b = a--`	`S1: b = a`
		`S2: a = a-1`

EXAMPLE

Consider the following code snippet:

```
int a = 5;
int b = a++; //a=5
int c = a;   //a=6
int d = ++a; //a=7

System.out.println("a: "+a+", b:
"+b+", c: "+c+", d: "+d);
```

Output:

```
a: 7, b: 5, c: 6, d: 7
```

Similarly for pre & post decrement operators:

```
int a = 5;
int b = a--; //a=5
int c = a;   //a=4
int d = --a; //a=3

System.out.println("a: "+a+", b:
"+b+", c: "+c+", d: "+d);
```

Output:

```
a: 3, b: 5, c: 4, d: 3
```

10. Arithmetic Exception

CONCEPT

When any numeric value is divided by **`0`**, we get an **`ArithmeticException`** following which the code terminates.

EXAMPLE

Consider the following code snippet:

```
int val = 5/0;

System.out.println(val);
```

Output:

```
java.lang.ArithmeticException: / by
zero
```

Table of Contents

1. Relational Operators

CONCEPT

Relational operators are used to compare values and return a boolean value.

```
1. > (greater than)
2. < (less than)
3. >= (greater than equal to)
4. <= (less than equal to)
5. == (equal to)
6. != (not equal to)
```

EXAMPLES

```
int val1 = 5, val2 = 8;
```

Expression	Result (ans)
`boolean ans = val1 > val2`	`false`
`boolean ans = val1 < val2`	`true`
`boolean ans = val1 >= val2`	`false`
`boolean ans = val1 <= val2`	`true`
`boolean ans = val1 == val2`	`false`
`boolean ans = val1 != val2`	`true`

NOTE

All relational operators can work on int and double but for **booleans** it only works with **==** and **!=**.

```
boolean val1 = true, val2 = false;
```

Expression	Result (ans)
`boolean ans = val1 > val2`	`Error`
`boolean ans = val1 == val2`	`false`

2. Logical Operators

CONCEPT

```
1. && (and)
2. || (or)
3. !  (not)
```

Logical operators operate on **`boolean`** data and they give result as a **`boolean`** value.

Operator	How does it work?
&&	**`true`** when both sides are true **`false`** otherwise
||	**`true`** when either or both sides are true, **`false`** otherwise
!	Works on one operand only and gives its opposite. **`!false`** is **`true`**

NOTE

If an expression contains arithmetic, relational and logical operators then the order of evaluation is as follows :
`arithmetic > relational > logical.`

3. Expression Equivalence

CONCEPT

Expression	Equivalent expression
`!(a&&b)`	`!a || !b`
`!(a||b)`	`!a && !b`
`!a || a`	`Always true`
`!a && a`	`Always false`
`a && (b||c)`	`(a&&b) || (a&&c)`

Expression	Equivalent expression
`a || (b&&c)`	`(a||b) && (a||c)`

*Given a, b and c are **boolean** values*

Expression	Equivalent Expression
`!(a<b)`	`a >= b`
`!(a<=b)`	`a > b`
`!(a==b)`	`a != b`

*Given a & b are **integer** values*

4. &&, || Short Circuiting

CONCEPT

Operator	Short circuiting operation
&&	Does not evaluate the RHS if the LHS is false
||	Does not evaluate the RHS if the LHS is true

EXAMPLES

```
int a = 2, b = 5;
boolean ans = a > b && b != 10
```

The LHS of `&&` evaluates to `false` and the right hand side therefore is not evaluated.

```
int val = 5;
boolean ans = ++val < 100 || ++val < 20;
```

`++val` is evaluated first. `val` becomes `6`.

```
boolean ans =  6 < 100 || ++val < 20;
```

Since `6` is less than `100` the condition is `true`. It short circuits and doesn't perform the operation of `++val<20`. Final value of `val` is `6` and `ans` is `true`.

5. If Statements

CONCEPT

```
if(<boolean expression/variable>)
    <stmt>
```

`<stmt>` is executed only when the condition evaluates to `true`.

EXAMPLES

Example 1:

```
int val = 2;
if(val>2)
   System.out.println("Hi");
//Output:
//There will be no output
```

Condition evaluates to `false` and hence no output.

Example 2:

```
int val = 2;
if(val>2)
   System.out.println("Hi");
   System.out.println("Always");
//Output:
//Always
```

Only the execution of the 1st statement is based on the `if`. The 2nd statement has therefore nothing to do with the `if` and will be executed.

Example 3:

```
int val = 2;
if(val>2){
   System.out.println("Hi");
   System.out.println("Always");
}
//Output:
//No output
```

The { } are used to associate multiple statements with the `if`. Since the condition is `false`, there will be no output.

Example 4:

```
int val = 12;
if(val>2){
   System.out.println("Hi");
   System.out.println("Always");
}
//Output:
//Hi
//Always
```

6. If-Else Statements

CONCEPT

```
if(<boolean expression/variable>)
```

```
        <stmt 1>
else
        <stmt 2>
```

`<stmt1>` will be executed `if` the condition is `true` otherwise `<stmt2>` will be executed.

EXAMPLES

Example 1:

```
int val = 2;
if(val>2)
   System.out.println("A");
else
   System.out.println("B");
//Output:
//B
```

Since the condition is not satisfied it goes to the `else` part of the code.

Example 2:

```
int val = 12;
if(val>2)
   System.out.print("A");
else
   System.out.print("B");
   System.out.print("C");
//Output:
//AC
```

Only 1 statement is associated with the `else`. Hence C will be printed irrespective of `val`'s value.

Example 3:

```
int val = 12;
if(val>2)
   System.out.print("A");
else {
   System.out.print("B");
   System.out.print("C");
}
//Output:
//A
```

`{ }` can be used to associate multiple statements with the `else` too.

Example 4:

```
int val = 12;
if(val>12)
   System.out.print("A");
else if(val>5)
   System.out.print("B");
else
   System.out.print("C");
//Output:
//B
```

`val>12` is checked first. Only **if the condition is not** satisfied, then `val>5` is checked. If that too is not satisfied then it goes to the `else`. In this case, `val>12` is not satisfied but `val>12` is.

Example 5:

```
int val = 14;
if(val>12)
   System.out.print("A");
else if(val>5)
   System.out.print("B");
else
   System.out.print("C");
//Output:
//A
```

Since the first `if` condition is satisfied, it won't go ahead and check the other condition since the checking of `val>5` happens only when the `val>12` is not satisfied.

Example 6:

```
int val = 14;
if(val>12)
   System.out.print("A");
if(val>20)
   System.out.print("B");
else
   System.out.print("C");
//Output:
//AC
```

The first `if` condition has got nothing to do with the second `if` condition. The `else` is associated only with the 2nd `if`. Hence first `val>12` will be verified. In this case, since it is `true` A will be printed. Next, `val>20` will be verified. Since, it's not true, then it goes to the `else` and prints C.

Example 7:

```
int val = 20;
boolean ans = val>14;
if(ans)
   System.out.println("Hi");
//Output:
//Hi
```

You can directly use a `boolean` in an `if`. If the variable is `true`, then the statement(s) associated with the `if` will be printed.

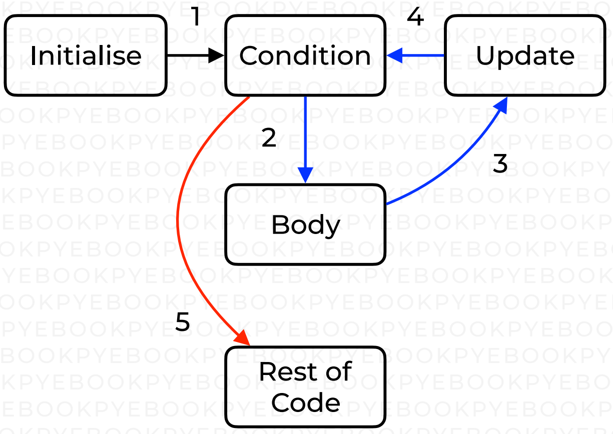
Table of Contents

1. for Loop

CONCEPT

```
for(<Initialisation>;<Condition>;<Update>)
    <Body>
```

Repeats the body till the condition doesn't come `false`. The condition needs to generate a `boolean`. The code flows like as shown below:



EXAMPLES

Example 1:

```
for(int i = 1; i <= 3; i++)
    System.out.print(i+ " ");
```

Output:

```
1 2 3
```

`i` is first given the value `1`, since the condition is `true`, it goes to the body and prints `1` and then goes to the update. Becomes `2` and checks the condition again and repeats till `i` becomes `4`.

Example 2:

```
for(int i = 1; i <= 3; i++)
    System.out.print("Hi ");
    System.out.print("Ok");
```

Output:

```
Hi Hi Hi Ok
```

Only 1 statement is associated with the `for`. **`Hi`** will be printed `3` times, it comes out of the loop and prints `Ok`.

Example 3:

```
for(int i = 1; i <= 2; i++){
    System.out.print("Hi ");
    System.out.print("Ok ");
}
```

Output:

```
Hi Ok Hi Ok
```

Both the statements are now part of the `for` loop's body.

2. while Loop

CONCEPT

After the initialisation, keeps repeating the body till the condition becomes false. Like for, just 1 statement is associated with the `while`.

```
<initialisation>
while (<condition>)
    <body> (update is a part of the body)
```

EXAMPLES

Example 1:

```
int val = 1;
while (val<5)
    System.out.print("Hi ");
```

Output:

```
Hi Hi Hi . . . ∞ printing infinitely
```

`val` is initially `1` and the condition is checked. Since `true`, `Hi` gets printed and it goes back to the condition. `val` is still `1` and has not been updated. Hence it becomes an **infinite loop**.

Example 2:

```
int val = 1;
while(val<5){
   System.out.print("Hi ");
   val++;
}
System.out.println(val);
```

Output:

```
Hi Hi Hi Hi 5
```

`val` is set to `1` and condition is checked. Since `true`, it enters the body. It prints `Hi` and increases `val` to `2` and checks the condition again. Repeats the process till the condition is not correct. When `val` becomes 5, the condition is `false` and it comes out of the loop and prints 5.

3. Jump Statements

CONCEPT

Jump statement	Working
`break`	Takes the control out of the loop
`continue`	In a for loop, skips rest of the body and goes to the update
	In a while loop, skips rest of the body and goes to the condition

EXAMPLES

Example 1:

```
for(int i = 1; i<=3; i++) {
   if(i==2) {
      break;
   }
   System.out.println(i);
}
```

Output:

```
1
```

Example 2:

```
int val = 4;
while(true){
   if(val ==7)
      break;
   System.out.print(val+" ");
   val++;
}
```

Output:

```
4 5 6
```

Example 3:

```
for(int i = 1; i<=3; i++) {
   if(i==2){
      continue;
   }
   System.out.print(i+" ");
}
```

Output:

```
1 3
```

4. Nested Loops

CONCEPT

Any type of loop can be nested inside any type of loop. The inner loop will be executed as many times as the outer loop condition is true.

EXAMPLE

Consider the following code snippet:

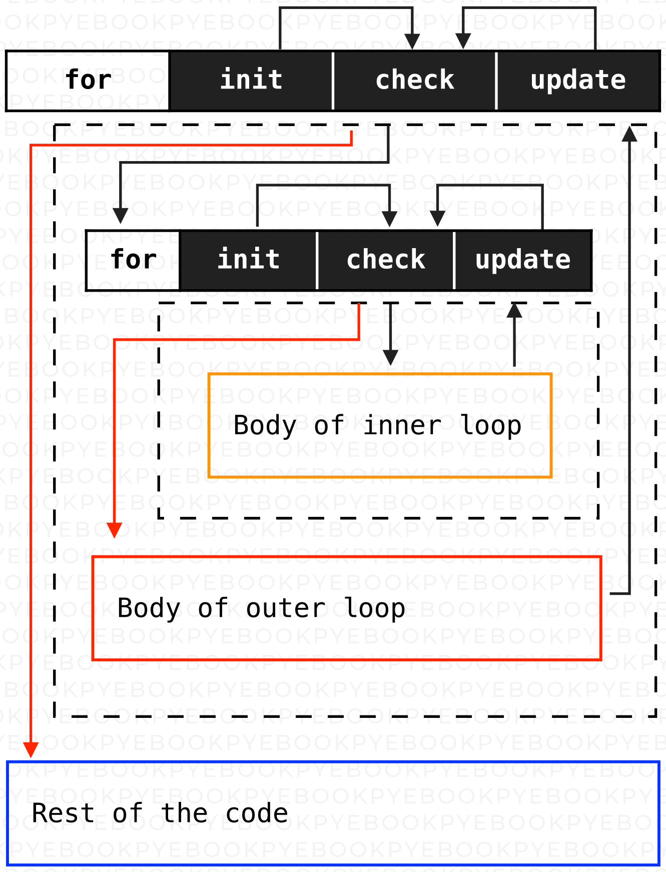
```
for(int i = 1; i <= 2; i++) {
   for(int j = 1; j <= i; j++) {
      System.out.print("* ");
   }
}
```

Output:

```
* * *
```

NOTE

Any jump statements placed inside the inner loop, will be applicable to that loop only. A `break` inside the inner loop will NOT take you out of the outer loop.



Flow of nested loop

5. Some Standard Loop-Based Algorithms

CONTINUOUS SUM

CONCEPT & CODE

Code below helps to find the sum of the numbers from `1` to `10`.

The logic is to take a variable and **keep accumulating in it.**

Code snippet:

```
int sum = 0;
for(int i = 1; i <= 10; i++)
   sum = sum + i; // This is the
important line
```

COUNTING OCCURRENCES

CONCEPT & CODE

Code below counts the number of factors of a number.

The logic is initialise **a count to `0` and keep adding `1`** when you encounter a factor.

Code snippet:

```
int count = 0, num;
//Assume num has been initialised
for(int i = 1; i <= 10; i++)
   if(num % i == 0)
      count = count + 1; //Can also
be count++
```

CHECKING THE PRESENCE OF A CERTAIN OCCURRENCE

CONCEPT & CODE

Code blow helps in checking whether a number is prime or not.

The logic is to take a variable and initialise it with a value. If there is any occurrence which is opposite of the desired occurrence, change the value of the variable. Once the loop completes check the status of the variable.

Code snippet:

```
boolean flag = true;
for(int i = 2; i < num; i++) {
   if(num%i == 0){
      flag = false;
   }
}
if(flag)
   System.out.print(“Prime”);
else
   System.out.print(“Not Prime”);
```

SUM OF THE DIGITS OF A NUMBER

CONCEPT & CODE

Code below finds the sum of the digits of a number irrespective of the number of digits.

The logic is extract the last digit and keep reducing the number by dividing it by `10` till the number reaches `0`.

Code snippet:

```
int num;
//Assume num has been initialised
while(num!=0){
   sum = sum + num % 10;
   num = num / 10;
}
```

Table of Contents

1. Basics

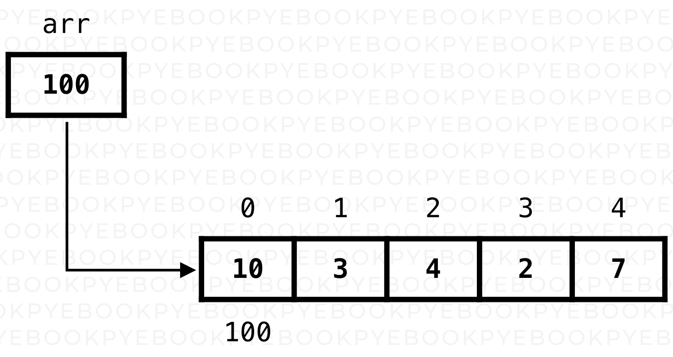
CONCEPT

An array is a collection of data of a similar type where the data is arranged in a contiguous memory location.

In general an array is declared as:

```
<data type> [] <name of array>; or
<data type> <name of array>[];
```

For the array below,
`int arr[] = {10, 3, 4, 2, 7};`



Memory representation of `arr`.

Some important points about arrays

1. Array names are references
2. References are locations in the memory
3. Positions begin from `0` and not `1`

For, the array above we assumed the memory location to be at 100, in reality it can be anything and is allocated automatically.

IMPORTANT ARRAY OPERATIONS

Consider the following array:
`int arr[] = {10, 3, 4, 2, 7};`

A. LENGTH PROPERTY

CONCEPT & EXAMPLE

Code snippet:

```
System.out.println(arr.length);
```

Output:

```
5
```

The length property gives the number of values in the array.

B. ACCESSING A VALUE IN AN ARRAY

CONCEPT & EXAMPLE

Code snippet:

```
System.out.println(arr[0]);
```

Output:

```
10
```

Positioning starts from `0` **and goes till** **`arr.length - 1`.**

Code snippet:

```
System.out.println(arr[5]);
```

Output:

```
ArrayIndexOutOfBoundsException
```

You get **`ArrayIndexOutOfBoundsException`** if you access a position that does not exist.

2. Creation of a New Array

CONCEPT

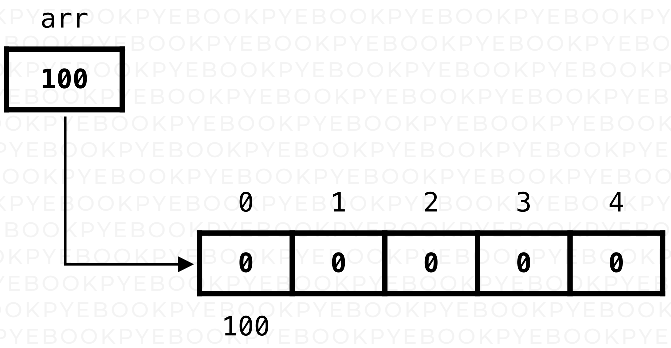
```
<data type> <array name> [] = new <data
type> [<size of array>];
```

NOTE

1. All values in an **`integer array`** are **`0`**
2. All values in a **`double array`** are **`0.0`**
3. All values in a **`boolean array`** are **`false`**

EXAMPLE

```
int arr [] = new int [5];
```



Memory representation of ***`arr`***

3. Iterate Through an Array

Consider the following array:
`int arr[] = { 1, 5, 2, 4, 6};`

ACCESS POSITION BY POSITION

CONCEPT & EXAMPLE

Code snippet:

```
for(int i = 0; i < arr.length; i++)
   System.out.print(arr[i] + " ");
```

Output:

```
1 5 2 4 6
```

ACCESS VALUE BY VALUE

CONCEPT & EXAMPLE

```
for(int val:arr)
   System.out.print(val + " ");
```

Output:

```
1 5 2 4 6
```

This loop is also called as the **enhanced loop or the for each loop**.

By Position	By Value
Can be used to modify the array	Cannot be used to modify the array
Can traverse through the array in a way e.g.access alternate values, go through the array in reverse etc.	Can only traverse from the beginning till the end of the array without skipping any value

Difference between the two ways of array traversal

4. Some Important Array Algorithms

LINEAR SEARCH

CONCEPT

Using the usual for loop:

```
int arr[] = //Array initialised
int val = //Value to be searched

boolean flag = false;

for(int i = 0; i<arr.length; i++) {
   if(arr[i] == val)
      flag = true;
}

if(flag)
   System.out.println("Present");
else
   System.out.println("Not
present");
```

Using the enhanced loop:

```
int arr[] = //Array initialised
int val = //Value to be searched
boolean flag = false;
for(int v: arr) {
   if(v == val)
      flag = true;
}
if(flag)
   System.out.println("Present");
else
   System.out.println("Not
present");
```

Alternate logic

Count the number of times val is present in the array. If the final count is 0 (check outside the loop), then the value is not present.

NOTE

A common **mistake** made by the students is to have an if-else inside the loop. The following code is **NOT CORRECT**

```
for(int i = 0; i<arr.length; i++) {
   if(arr[i] == val)
      System.out.println("Present");
   else
      System.out.println("Not Present");
}
```

FINDING MAX VALUE IN AN ARRAY

CONCEPT

Using the usual for loop:

```
int arr[] = //Array initialised
int max = arr[0];
for(int i = 0; i<arr.length; i++) {
   if(arr[i] > max)
      max = arr[i];
}
System.out.println(max);
```

Using the enhanced for loop:

```
int arr[] = //Array initialised
int max = arr[0];
for(int val: arr) {
   if(val> max)
      max = val;
}
System.out.println(max);
```

Alternate logic

Instead of initialising max to arr[0], you can also initialise max to Integer.MIN_VALUE

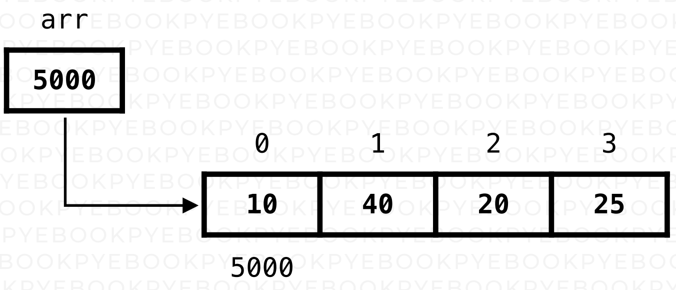
NOTE

A common mistake made by the students is to give an initial value of max as 0. It won't work if the values in the array are all negative.

5. Working With Array References

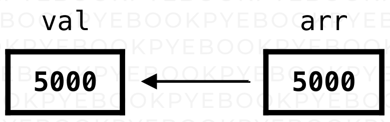
Consider the code

`int arr[] = {10,40,20,25};`

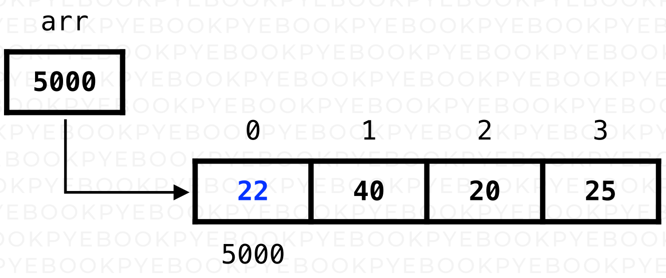


Memory representation of ***arr***

`int val[] = arr;`



`arr[0] = 22;`



Updated memory representation of ***arr***

`System.out.println(val[0]);`

Output: **22**

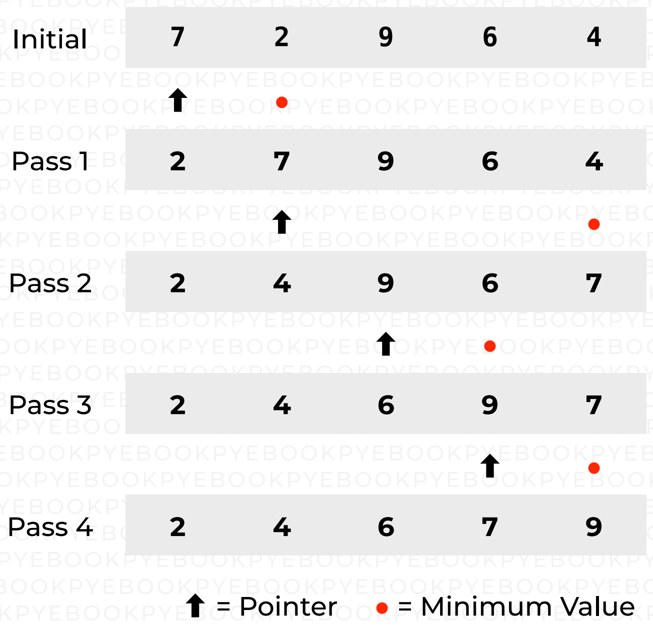
6. Sorting and Searching Algorithms

SELECTION SORT

ALGORITHM

1. Stand at the `0th` position
2. Assume the `0th` position value to be minimum & find the minimum in the remainder of the array
3. Swap the values of the `0th` position with the position where the minimum was found
4. Repeat steps `1 to 3` for all the other positions of the array

```
int arr[] = //Some values;
for (int i = 0; i < arr.length-1; i++) {
// Find the minimum element in unsorted
array
   int min_idx = i;
   for (int j = i+1; j < arr.length; j++) {
      if (arr[j] < arr[min_idx])
         min_idx = j;
   }
// Swap the found minimum element with the
first element
   int temp = arr[min_idx];
   arr[min_idx] = arr[i];
   arr[i] = temp;
}
```



ANALYSIS

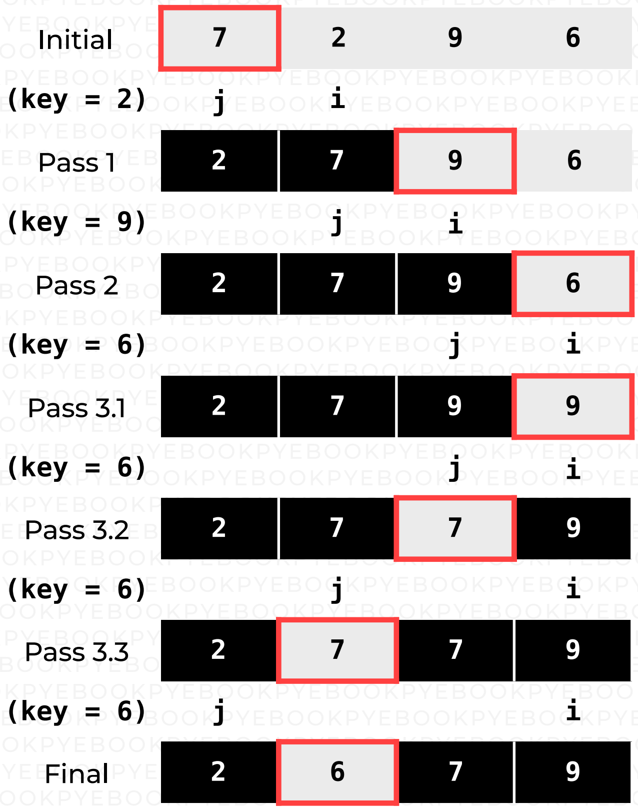
1. If there are n values in the array. The first time the inner loop will execute n-1 times, the second time it will execute n-2 times, and so on.
2. This algorithm runs the same number of times irrespective of the original arrangement of the values of the array.
3. The inner loop runs a total of n-1 times.

INSERTION SORT

ALGORITHM

1. Start from position `1` (say `i`).
2. Store the value at position `1` in a `temp` variable and start from position `0`.
3. Compare the value at position `0` (say `j`) with `temp` and if the value is greater move the value to the right. Keep decreasing `j` till it doesn't reach `−1` or the value doesn't becomes lesser than `temp`.
4. Place the `temp` value at `j + 1` and repeat steps 2 - 3 for an increased value of `i`.

```
int arr[] = // Some values;
for (i = 1; i < arr.length; i++) {

   key = arr[i];
   j = i − 1;

   while (j >= 0 && arr[j] > key) {
      arr[j + 1] = arr[j];
      j = j − 1;
   }

   arr[j + 1] = key;
}
```



ANALYSIS

1. If the original array is in descending order and has `n` values then the inner loop for the first time will execute `1` time and then will execute `2` times and so on. When it reaches the last value, the inner loop will execute `n−1` times.
2. If the original array is in increasing order, then the inner loop won't execute at all since the `while` condition is always `false`.

BINARY SEARCH

ALGORITHM

NOTE
The array must be in a sorted order (increasing or decreasing). The algorithm below is written for an increasing order array.

1. Initialise `low` to `0` and `high` to `mid - 1`.
2. Calculate `mid` as `(low+high)/2`.
3. Compare `arr[mid]` with `val`. If the `val` is more than `arr[mid]` then change your `low` to `mid+1`. If it's less than `arr[mid]` then change the `high` to `mid–1`.If its the same as `arr[mid]` you stop the process.
4. Keep doing the process till `low <= high`. If the `low` becomes more than `high` then the value doesn't exist.

```
int low = 0, high = arr.length - 1;
while (low <= high) {
    int m = (low + high) / 2;
    // Check if x is present at mid
    if (arr[m] == x)
        return m;
    // If x greater, ignore left half
    if (arr[m] < x)
        low = m + 1;
    // If x is smaller, ignore right
    half
    else
        high = m - 1;
}
// if we reach here, then element
was not present
return -1;
```

Index	0	1	2	3	4
Find 4	2	4	6	7	9
	↓				↑
			■		
`arr[m] > x`	↓	↑			
	■				
`arr[m] < x`		↓ ↑			
		■			
`arr[m] = x` *Value Found*		m = 1			

↑ = High ↓ = Low ■ = Mid

ANALYSIS

1. Binary search on average is faster than linear search. It can perform a search in $\log_2 n$ where n is the number of values to be searched.
2. Linear search is the fastest if the value to be searched is at the first position and binary search is the fastest if the value to be searched is in the middle of the array.

Table of Contents

1. What Is a Class?

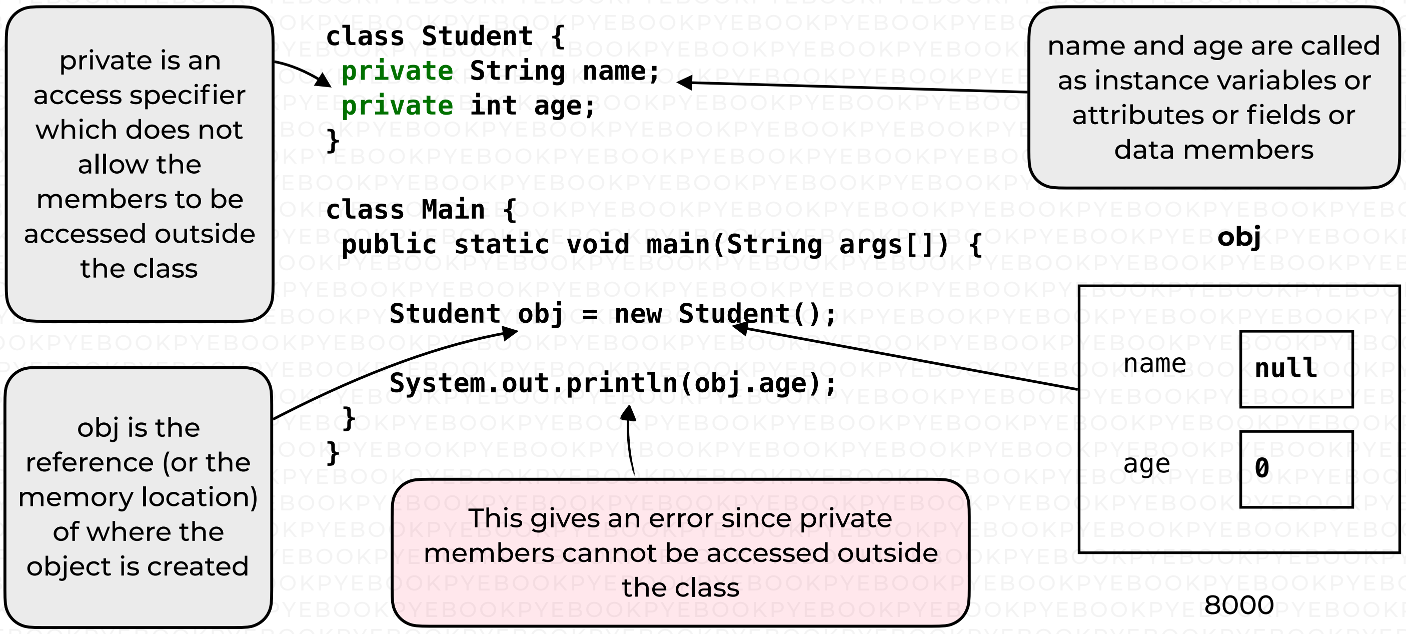
CONCEPT

`Class` is a logical name given to a collection of attributes usually of different data types. A `class` is also called a secondary data type or a user-defined data type.

2. What Is an Object/Instance of a Class?

CONCEPT

An object is the actual manifestation of the class. The class defines the blueprint whereas the object is the actual creation which occupies memory as per the design specified by the class.



```
class Student {
 private String name;
 private int age;
}

class Main {
 public static void main(String args[]) {

    Student obj = new Student();

    System.out.println(obj.age);
 }
}
```

private is an access specifier which does not allow the members to be accessed outside the class

name and age are called as instance variables or attributes or fields or data members

obj is the reference (or the memory location) of where the object is created

This gives an error since private members cannot be accessed outside the class

3. Methods in a Class

CONCEPT & EXAMPLE

A method is a set of statements that performs a specific task. Usually, they are used to manipulate certain data members of the class. Unlike instance variables, methods are usually kept `public`.

General structure of a **method definition** in a class:

```
<Access Specifier> <Return Type> <Method Name> (<List of parameters separated by
comma>) {
  <Body of the method>
}
```

General structure of a **method call** outside the class:

```
<Instance of the class>.<method-name>(<List of parameters>)
```

What does the `return` type mean?
Return type is the data type of the answer returned by the method. If the method doesn't return any value, then the return type is `void`.

What do you mean by parameters?
Parameters are variables (apart from the instance variables) needed before the function starts executing its work.

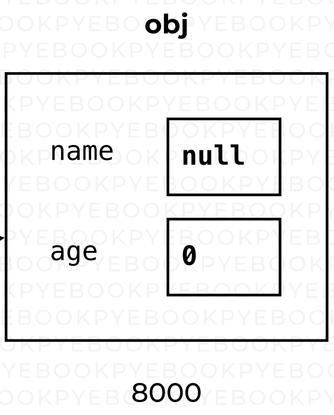
A. Return Type — `void` | Parameters — `none`

When the return type of the method is `void`, this means the method will not return any value.

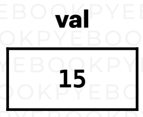
`public` is an access specifier which allows members to be accessible anywhere

```
class Student {
    private String name;
    private int age;

    public void showAge() {
        System.out.print(age);
    }
}

class Main {
    public static void main(String args[]) {
        Student obj = new Student();
        obj.showAge();
    }
}
```

Will output `0`

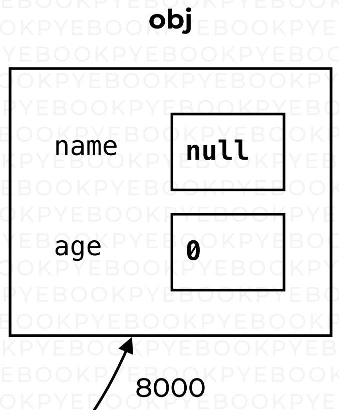


B. Return Type — void | Parameters — present

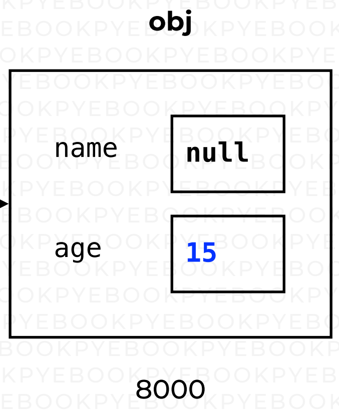


```
class Student {

    private String name;
    private int age;

    public void setAge(int val) {
        age = val;
    }

}

class Main {
    public static void main(String args[]) {

        Student obj = new Student();
        obj.setAge(15);
    }
}
```

setAge is **called as a mutator** since it sets the value of a member



After setAge is called



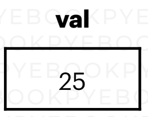
C. Return Type — present | Parameters — present

```
class Student {

    private String name;
    private int age;

    public void setAge(int val) {
        age = val;
    }

    public int getIncreasedAge() {
        int val = age + 10;
        return val;
    }

}
class Main {
    public static void main(String args[]) {

        Student obj = new Student();
        obj.setAge(15);

        int value = obj.getIncreasedAge();
    }
}
```

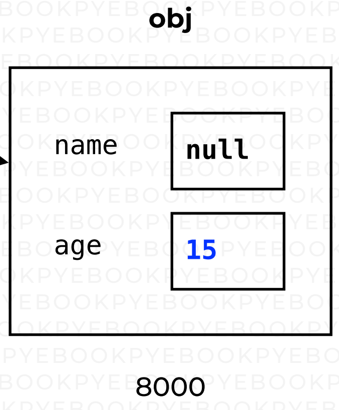
If a method has a return type, then it must return a value of that type **always**.

Here getIncreasedAge is also called as an accessor since it access a member of the class.

In its simplest form, an accessor just returns the value of a member

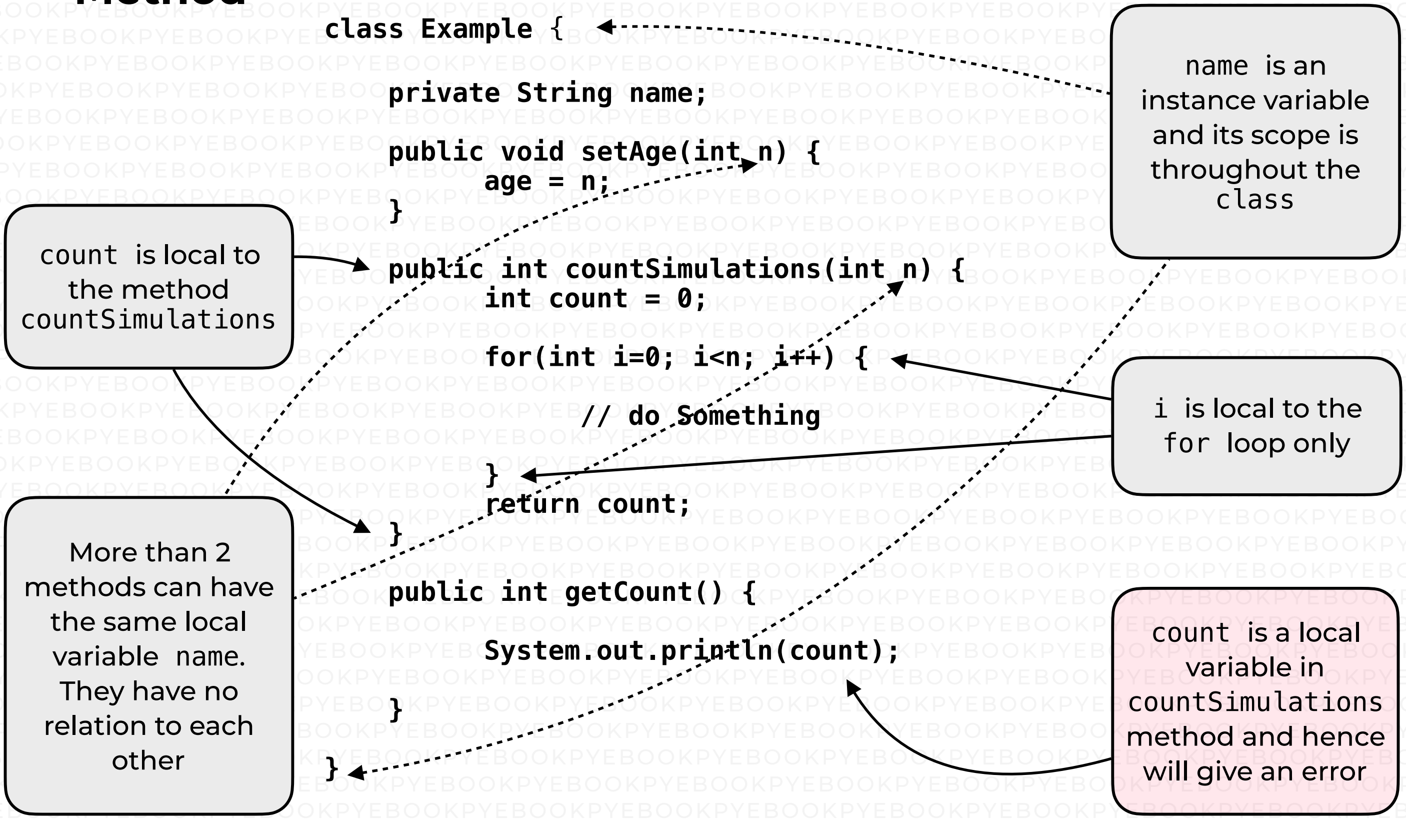


The state of the instance after the setAge is called.



value will have 25 which is the number returned by getIncreasedAge

4. Scope of the Variables Used Within a Class and Within a Method



```
class Example {
    private String name;
    public void setAge(int n) {
        age = n;
    }
    public int countSimulations(int n) {
        int count = 0;
        for(int i=0; i<n; i++) {
            // do Something
        }
        return count;
    }
    public int getCount() {
        System.out.println(count);
    }
}
```

name is an instance variable and its scope is throughout the class
count is local to the method countSimulations
i is local to the for loop only
More than 2 methods can have the same local variable name. They have no relation to each other
count is a local variable in countSimulations method and hence will give an error

5. Constructors in a Class

CONCEPT

1. The purpose of a constructor is to create the instance of the class and assign initial values to the instance members.
2. It has the same name as the class and does not have a return type.
3. While creating the object, the constructor always needs to be called with the new keyword.
4. For instance members of a primitive type, the default values are 0 for int, 0.0 for double and false for boolean. Whereas, secondary data types are initialised with the value null.
5. You can create multiple constructors in the same class. They need to differ with respect to the parameters.

EXAMPLE

```
class Student {

    private String name;
    private int age;

    public Student() {
        name = null;
        age = 0;
    }

    public Student() {
        name = “AB”;
        age = 20;
    }

    public Student(String n, int a) {
        name = n;
        age = a;
    }
}
class Main {
    public static void main(String args[]) {

        Student obj = new Student();

        Student param = new Student(“XY”,12);
    }
}
```

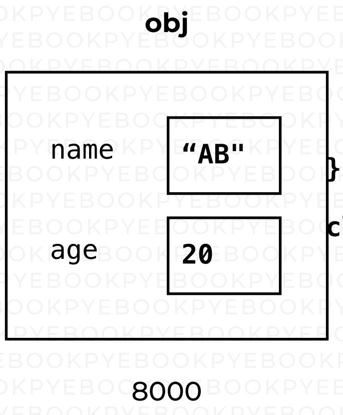
This is the default constructor Java includes in the class if you do not add one

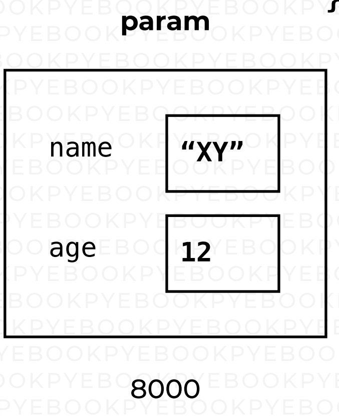
This default Constructor has no parameters.

This is constructor with parameters. You can have as many constructors as needed as long as they differ in parameters

Object creation using default constructor

Object creation using the parameterised constructor





6. Method and Constructor Overloading

CONCEPT

Constructors and methods can be overloaded. **Overloading** is the process of having multiple methods or constructors with the **same name but different parameters**. Difference in parameters can be made by:

1. Number of parameters
2. Data type of parameters
3. Order in which parameters are declared

EXAMPLE

Consider the `class Student` as seen below:

```
class Student {

    public void setInfo(int value) {
            // Does something
    }

    public void setInfo(String val) {
            // Does something
    }

    public void setInfo(int v, double r) {
            // Does something
    }
}
```

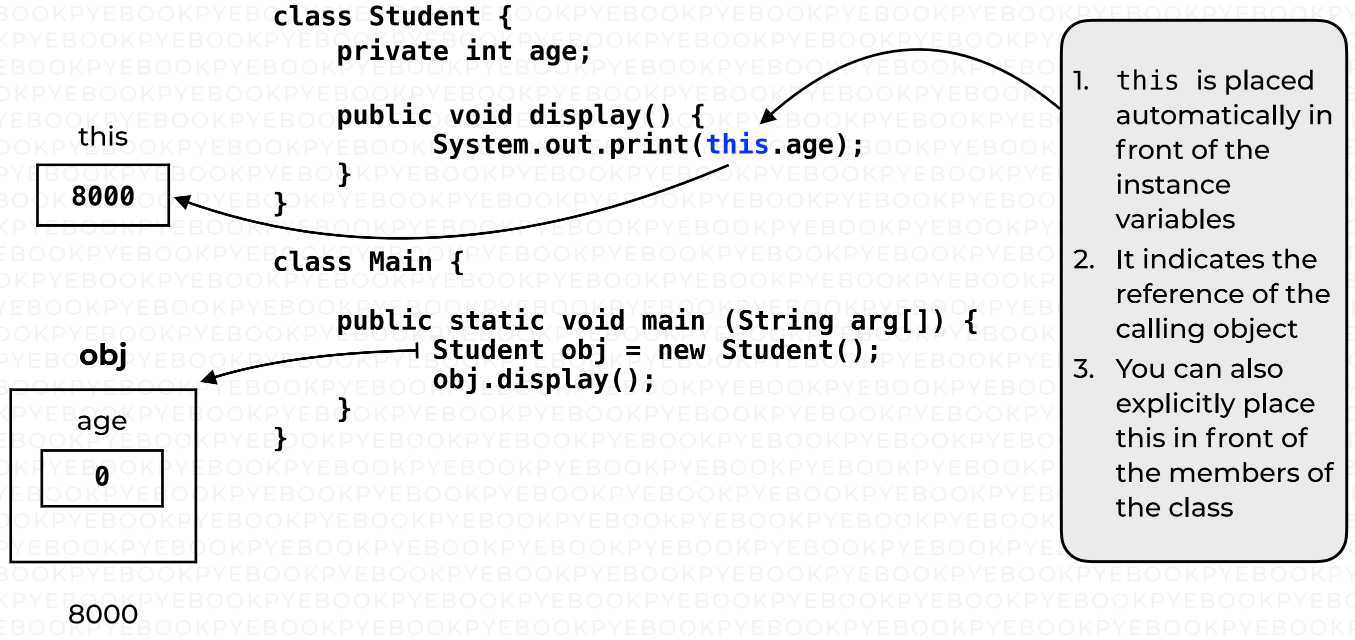
For the above class, here is a list of method declarations that will be allowed and that won't be:

Method	After Compiling	Reason
`public int setInfo(String n)`	Error	Return type being different than the existing methods in the class DOES NOT make it eligible for overloading.
`public void setInfo(String na, int ag)`	No Error	The number of parameters are different than the one in the class.
`public void setInfo(double na, int ag)`	No Error	The order of the parameters is different
`public void setInfo(int n)`	Error	There is already a `setInfo` method which takes an int as a parameter. The name of the parameter being different is

7. this Keyword

CONCEPT

The keyword `this` is **present automatically inside non-static methods** and refers to the **reference of the calling object.**



```
class Student {
    private int age;

    public void display() {
        System.out.print(this.age);
    }
}
class Main {

    public static void main (String arg[]) {
        Student obj = new Student();
        obj.display();
    }
}
```

1. `this` is placed automatically in front of the instance variables
2. It indicates the reference of the calling object
3. You can also explicitly place this in front of the members of the class

The keyword `this` needs to be **placed explicitly in methods and constructors, if local variables and/ or parameters have the same name as the instance members.**

```
class Student {

   private int age;

   public Student (int age) {
       this.age = age;
   }
}
```

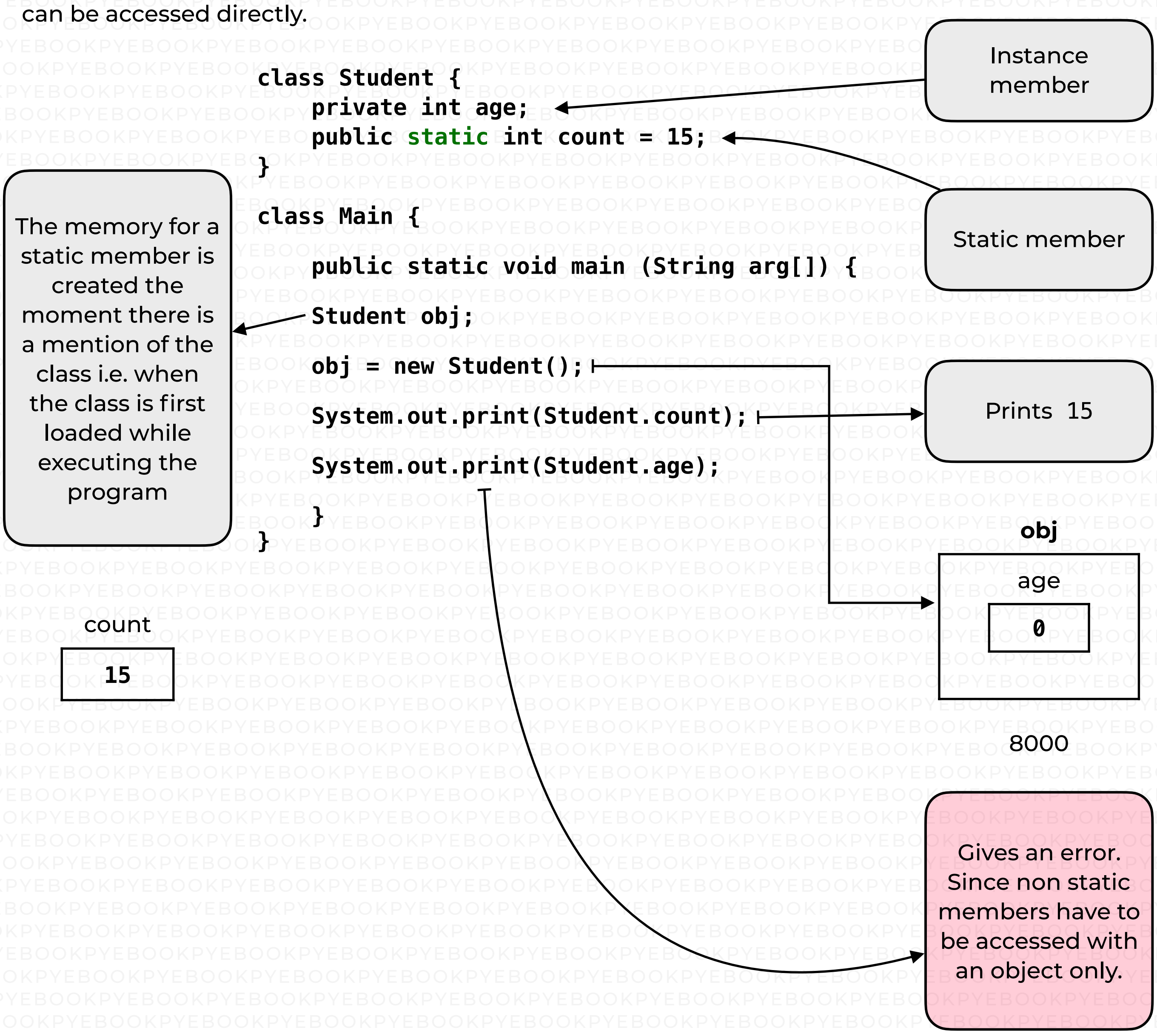
If there is an instance variable and a local variable with the same name, then the preference is given to the local variable and hence `this.` should be placed to refer to the instance variable

8. Static Members and Methods

STATIC DATA MEMBERS

CONCEPT

1. Instance members belong to the object and hence the number of copies depends on the number of instances created.
2. Static data members, only 1 copy is created.
3. All the instances share the same copy of the variable.
4. If possible(i.e. if it is public) it needs to be accessed with the class name. Within the same class it can be accessed directly.



```
class Student {
    private int age;
    public static int count = 15;
}

class Main {

    public static void main (String arg[]) {

        Student obj;

        obj = new Student();

        System.out.print(Student.count);

        System.out.print(Student.age);

    }
}
```

Instance member

Static member

The memory for a static member is created the moment there is a mention of the class i.e. when the class is first loaded while executing the program

Prints 15

Gives an error. Since non static members have to be accessed with an object only.

STATIC METHODS

CONCEPT

1. Usually used to work with static members
2. Since it belongs to the class, it does not have the `this` keyword inside it
3. Therefore non static or instance members can't be accessed **directly** inside it
4. Static members can be accessed in any method - static or non-static

EXAMPLE

```
class Student {

    private int age;
    public static int count = 15;

    public void display(){
        System.out.print (this.age);
        System.out.println(count);
    }

    public static void inc() {
        System.out.print(age);
        Student obj = new Student();
        obj.age = 20;
        count++;
    }
}
class Main {

    public static void main (String arg[]) {

    Student.inc();

    }
}
```

There is an object referencing age and hence this is allowed

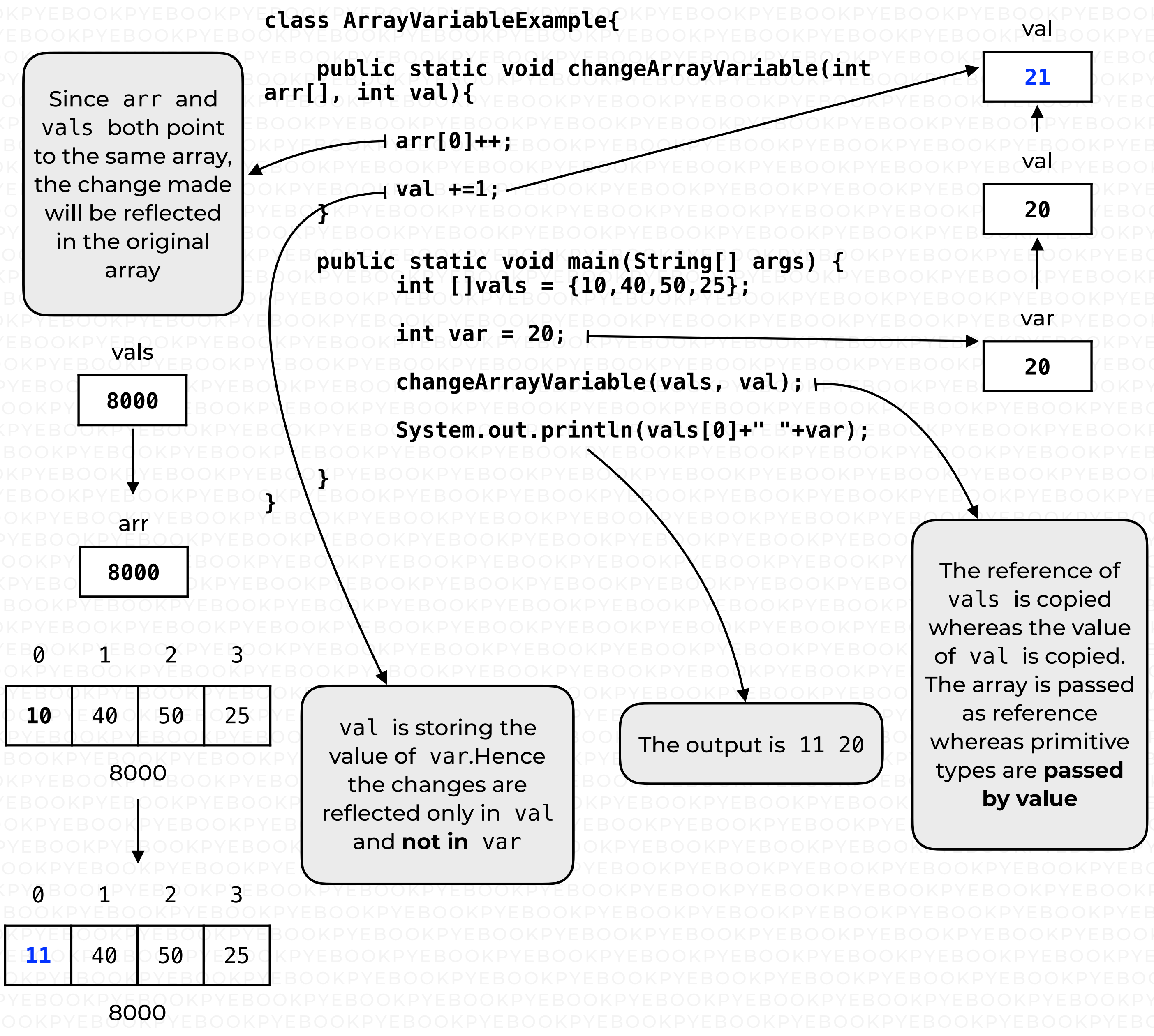
Static members can be accessed inside non static methods

Since there is no `this` inside a static method, the non static members cannot be accessed directly

Accessing a static method outside the class with the class name

9. Pass by Value & Pass by Reference

CONCEPT & EXAMPLE



```
class ArrayVariableExample{

    public static void changeArrayVariable(int
arr[], int val){
            arr[0]++;
            val +=1;
    }
    public static void main(String[] args) {
            int []vals = {10,40,50,25};

            int var = 20;
            changeArrayVariable(vals, val);
            System.out.println(vals[0]+" "+var);
    }
}
```

Since `arr` and `vals` both point to the same array, the change made will be reflected in the original array

`val` is storing the value of `var`.Hence the changes are reflected only in `val` and **not in** `var`

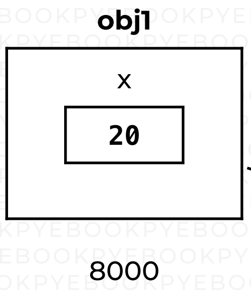
The output is 11 20

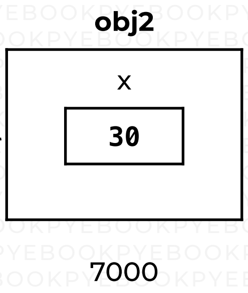
The reference of `vals` is copied whereas the value of `val` is copied. The array is passed as reference whereas primitive types are **passed by value**

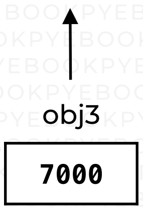
10. Working With References & Aliasing

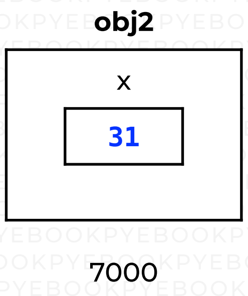
CONCEPT & EXAMPLE

```
class SomeNumber{
    private int x;

    public SomeNumber(int x){
        this.x = x;
    }

    public void incBy1(){
        this.x+=1;
    }

    public void display(){
        System.out.println(x);
    }

}

public class Main
{
    public static void main(String[] args) {

    SomeNumber obj1 = new SomeNumber(20);
    // Let's assume obj1's reference is 8000

    SomeNumber obj2 = new SomeNumber(30);
    // Let's assume obj2's reference is 7000

    SomeNumber obj3 = obj2;

    obj2.incBy1();

    obj3.display();
    }
}
```









Will display **31** since `obj2` is the same as `obj3`

It increases the value of `x` at location `7000` from `30` to `31`

`obj3` is an alias of `obj2`. Which means whatever object `obj2` is pointing at, `obj3` will have the same reference as `obj2`

11. Math Class

All members and methods of the **Math class are static** and hence referenced by the name of the class.

`1. public final static PI`

CONCEPT & EXAMPLE

`final static` data member of the Math class. The value is value of the mathematical notation π.

Example:

```
int radius = 4;
double area = Math.PI * radius *
radius // Calculates area of circle
```

`2. public static int abs(int x)`

CONCEPT & EXAMPLE

Static method which returns the absolute value(positive value) of an `integer`.

Example:

```
int y = -4;
int ans = Math.abs(y);
```

`ans` will have the value:

```
4
```

`3. public static double abs(double x)`

CONCEPT & EXAMPLE

Static method which returns the absolute value(positive value) of a `double`.

Example:

```
double y = -5.2;
double ans = Math.abs(y);
```

`ans` will have the value:

```
5.2
```

`4. public static double pow(double a, double b)`

CONCEPT & EXAMPLE

Returns a^b.

Example:

```
int y = 2, z = 3;
double ans = Math.pow(y, z);
```

`ans` will have the value:

```
8.0
```

`5. public static double sqrt(double val)`

CONCEPT & EXAMPLE

Returns the positive square root of the number.

Example:

```
int y = 4;
double ans = Math.sqrt(y);
```

Output:

```
2.0
```

`6. public static double random()`

CONCEPT & EXAMPLE

Returns a random **between 0.0 (included) and 1.0 (not inclusive)**.

Example:

```
double val = Math.random();
```

`val` can be:

```
0.12
```

NOTE

`random()` can be used to generate a random integer between `n1` and `n2` (where `n1 < n2`)

```
int rand = (int)(Math.random()*(n2-n1+1))
+ n1;
```

12. String Class

A `String` (a secondary data type) is an object of the String class which denotes **a collection of one or more characters enclosed in quotation marks.**

DIFFERENT WAYS OF CREATING STRINGS

CONCEPT

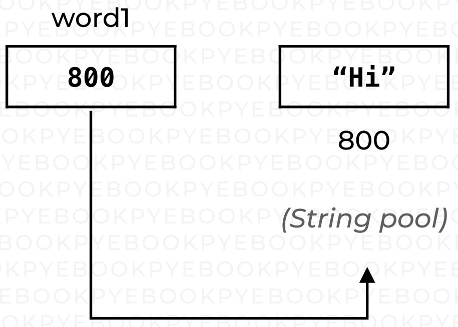
Using the quotes	Using the constructor
`String word1 = “Hello”;`	`String word2 = new String(“Hello”);`
Strings are created in the String pool	Strings are created outside the String pool
Inside the string pool, no duplicates can exist	Outside the string pool, duplicates can exist

NOTE

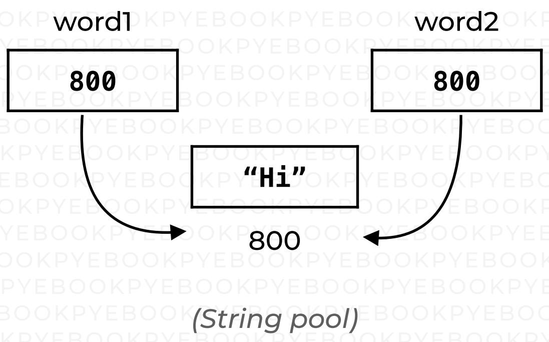
The name of the `String` **is a reference** of where the String is stored.

EXAMPLES

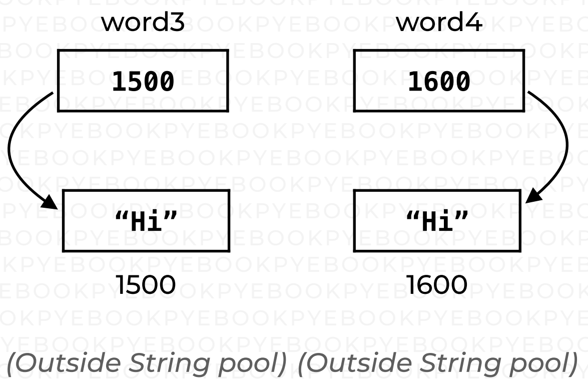
```
String word1 = “Hi”;
// word1 is reference to “Hi”
```



```
String word2 = “Hi”;
// word2 references to the same “Hi”
since duplicates can’t exist in the pool
```

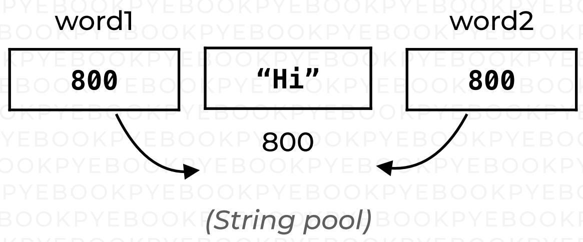


```
String word3 = new String(“Hi”);
String word4 = new String(“Hi”);
// Both word3 and word4 are outside the
pool and are at different locations
```



For the above set of Strings created (the memory allocation is shown above).

```
boolean ans1 = word1 == word2;
```



The value of `ans1` here will be:

```
true
```

```
boolean ans2 = word1 == word3;
```

The value of `ans2` here will be:

```
false
```

```
boolean ans3 = word3 == word4;
```

The value of `ans3` here will be:

```
false
```

NOTE

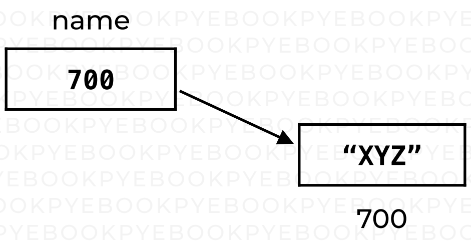
Here the references are being compared and not the contents.

STRING IMMUTABILITY

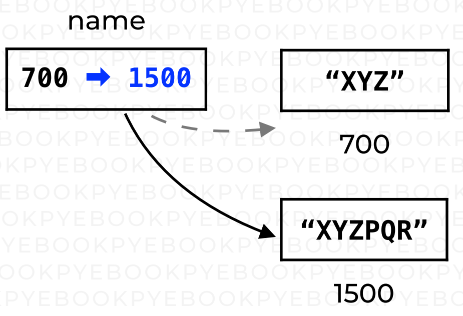
CONCEPT & EXAMPLE

Once a string is created at a particular location, **at that location the contents of the String can't be changed.**

```
String name = “XYZ”;
```



```
name = name + “PQR”;
```



What you saw was when you made the change to name, it actually created a new memory location with the new String and the reference of the name changed.

NOTE

Whenever Strings are modified, a new String is created. When Strings are passed to methods and the methods modify the String, then the String in the calling function remains unchanged.

STRING METHODS

1. `public int length()`

CONCEPT & EXAMPLE

Returns the length of the String object.

Example:

```
String word = “APCSA”;
int len = word.length();
```

`len` will have the value :

```
5
```

2. `public String substring(int startPos)`

CONCEPT & EXAMPLE

Returns the part of the string starting from `startPos` till the end. Remember positioning begins from 0.

Example:

```
String word = “APCSA”;
String part = word.substring(2);
```

`part` will have the value:

```
CSA
```

3. `public String substring(int startPos, int endPos)`

CONCEPT

Returns the part of the string starting from `startPos` **till the `endPos` not including `endPos`.**

EXAMPLE

Example 1:

```
String word = “APCSA”;
String part = word.substring(2,4);
```

`part` will have the value:

```
CS
```

Example 2:

```
String word = “APCSA”;
String part = word.substring(4,5);
```

`part` will have the value:

```
A
```

Example 3:

```
String word = “APCSA”;
String part = word.substring(5,6);
```

The output will be:

```
StringIndexOutOfBoundsException
```

4. `public boolean equals(Object other)`

CONCEPT & EXAMPLE

Returns `true` if the current string is exactly the same as the other string otherwise `false`.

Example:

```
String word1 = “AP”;
String word2 = new String(“AP”);
String word3 = “ap”;
boolean ans = word1.equals(word2);
boolean ans2 = word1.equals(word3);
System.out.print(ans + “ “ + ans2);
```

The output will be:

```
true false
```

5. `public int indexOf(String part)`

CONCEPT & EXAMPLE

Returns the position of the 1st occurrence where the `part` is in the current string. If the part doesn’t exist then it’s `-1`.

Example:

```
String word = “APCSA”;
int val = word.indexOf(“A”);
int val2 = word.indexOf(“PC”);
int val3 =word.indexOf(“PCT”);
System.out.print(val + ” “ + val2 +
“ “ + val3);
```

The output will be:

```
0 1 -1
```

6. `public int compareTo(String other)`

CONCEPT & EXAMPLE

Returns

1. **a `value < 0` if this is less than other**
2. **zero if this is equal to other**
3. **a `value >0` if this is greater than other**

Example:

```
String word1 = “APCSA”;
String word2 = “CS”;
int val = word1.compareTo(word2);
// -ve value indicating word1 comes
// alphabetically ahead of word2

int val2 = word2.compareTo(word1);
// +ve indicating word2 comes
alphabetically after word1

String word3 = “CS”;
int v = word2.compareTo(word3);
// gives 0
```

SOME IMPORTANT STRING ALGORITHMS

A. STRING COMPARISON

CONCEPT

If there are two Strings, in order to check whether they have the same contents or not use either **the `equals`** method or **the `compareTo`** method.

ALGORITHM

```
String word1 = “XYZ”;
String word2 = “PQR”;
if(word1.equals(word2))
   System.out.println(“Words are same”);
else
   System.out.println(“Words not same”);
```

NOTE

A very common mistake is to write the `if` condition as:

```
if(word1 == word2)
```

This **does not** compare the contents but instead compares the references.

B. ACCESSING EACH LETTER OF A STRING

ALGORITHM

We use substring to do the same.

```
String word = “APCSA”;
for(int i = 0 ; i <word.length(); i++)
   System.out.println(word.substring(i,
i+1));
```

C. CHECK PRESENCE OF A STRING/ CHARACTER IN ANOTHER STRING

CONCEPT

Use `indexOf` to check whether a String or a character is present in another String or not. If it’s not present indexOf will return –1

ALGORITHM

```
String word1 = “XYZ”;
String word2 = “PQR”;
if(word1.indexOf(word2) != -1)
   System.out.println(“word2 is present
in word1 ”);
else
   System.out.println(“word2 is not
present in word1);
```

D. CREATE A NEW STRING FROM AN EXISTING STRING

CONCEPT

If the requirement is to create a new String from another String, then start out with an empty String and keep appending to it. The code below creates a new string without the letter **a** in it.

ALGORITHM

```
String word = “data”;
String newword = “”;
for(int i = 0 ; i <word.length(); I++) {
   if(!word.substring(i,
i+1).equals(“a”)) {
      newword += word.substring(i, i+1);
   }
}
```

E. REPLACING THE OCCURRENCE OF THE OLD STRING WITH THE NEW STRING.

CONCEPT

Logic is as follows:

1. Store the original string in an another String.
2. Find the occurrence of old in the other String.
3. If present change the other string with the new string and go back to step 2. When there are no more occurrences left, then stop the process.

ALGORITHM

```
String input;
// Some input
String old;
// word to be replaced
String newWord;
// Word to be replaced with

String updated = input;
int found = updated.indexOf(old);
// find first location of old
while (found != -1) {
    // loop while old is still in the
String
    // replace old with newWord
    updated = updated.substring(0, found)
+ newWord + updated.substring(found +
old.length());
    found = updated.indexOf(old); // find
next occurrence;
}
System.out.println(updated);
```

Table of Contents

1. Basics of ArrayList & Drawbacks of Arrays

CONCEPT

For an array,

1. Size needs to be specified while array creation or the values need to be given directly.
2. Once created, the size cannot change.
3. If you need to modify the array, the only way is to create a new array with the modified size and copy the values from the original to the new array if needed.

`ArrayList` is a **dynamic data structure** which can alter its size on the fly and does not have to have a specific size mentioned at the time of creation. It is part of the `java.util` package.

ARRAYLIST CREATION

```
ArrayList<E> list = new ArrayList<E>();
```

NOTE

E mentioned above can **only be a secondary datatype** like `String`, or any other user defined class. It cannot be primitive data types like `int`, `boolean` and `double`.

EXAMPLE

Consider an ArrayList of Strings declared below.

```
ArrayList<String> names = new
ArrayList<String>();
```

For the below set of methods, `names` is the `ArrayList` which is used all throughout below

ARRAYLIST METHODS

`1. public boolean add(E obj)`

CONCEPT & EXAMPLE

Adds the **object to the end of the list**

Code snippet:

```
names.add("XYZ");
names.add("PQR");
```

Contents of **`names`** after the above code:

```
["XYZ", "PQR"]
```

`2. public int size()`

CONCEPT & EXAMPLE

Returns the **number of elements** in the list.

Code snippet:

```
int val = names.size();
```

Contents of **`val`** after the above code:

```
2
```

`3. public void add(int index, E obj)`

CONCEPT & EXAMPLE

Inserts obj at position index (`0 <= index <= size`), moving elements at position `index` and higher to the right (adds 1 to their indices) and adds 1 to size.

`names` list so far is:

```
["XYZ", "PQR"]
```

Now when we do,

```
names.add(1, "LMN");
```

Contents of **`names`** after the above code:

```
["XYZ", "LMN", "PQR"]
```

4. `public E get(int index)`

CONCEPT & EXAMPLE

Returns the element at position `index` in the list. Positioning begins from 0.

`names` list so far is:

```
["XYZ", "LMN", "PQR"]
```

For the code below:

```
String val = names.get(2);
```

Contents of **`val`** after the above code

```
PQR
```

5. `public E set(int index, E obj)`

CONCEPT & EXAMPLE

Replaces the element at position `index` with `obj`; returns the element formerly at `index`.

`names` list so far is:

```
["XYZ", "LMN", "PQR"]
```

For the code below:

```
String old = names.set(2, "PTY");
```

Contents of **old** after the above code:

```
PQR
```

Contents of **names** after the above code:

```
["XYZ", "LMN", "PTY"]
```

6. `public E remove(int index)`

CONCEPT & EXAMPLE

Remove element from `index`, moving elements at position `index + 1` and higher to the left (subtracts 1 from their indices) and subtracts `1` from size; returns the element formerly at position `index`.

`names` list so far is:

```
["XYZ", "LMN", "PTY"]
```

For the code below:

```
String old = names.remove(1);
```

Contents of **`old`** after the above code:

```
LMN
```

Contents of **`names`** after the above code

```
["XYZ","PTY"]
```

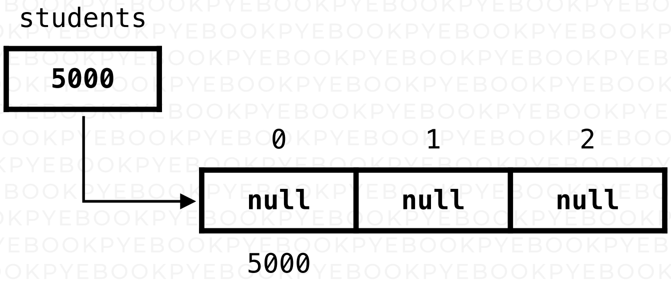
2. Working With Array of Objects

Consider the following piece of code.

```
class Student {
   private String name;
   private int age;

   public Student(String name, int age){
      this.name = name;
      this.age = age;
   }

   public int getAge(){
      return age;
   }

   public boolean isAdult(){
      return age > 18;
   }
}
```

CREATION OF ARRAY OF OBJECTS

```
Student students[] = new Student[3];
```



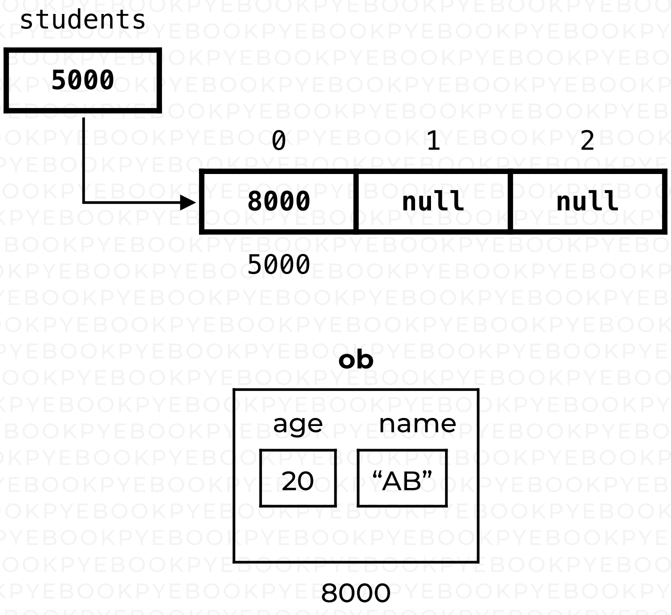
*Memory representation of **students***

`students` is a reference to an array of references. By default, the references are `null`.

ASSIGNING AN OBJECT IN THE ARRAY

We can create an object using the constructor and then allocate the reference to a certain array position.

```
students[0] = new Student("AB", 20);
```



*Memory representation of **students[0]***

NULL POINTER EXCEPTION WITH ARRAY OF OBJECTS

If we try to do the following,

```
boolean val = students[1].isAdult();
```

This will give you a `NullPointerException` since `students[1]` is still `null` and no Student object has been created yet.

TRAVERSING AN ARRAY OF OBJECTS

The code below counts the number of students that are adults.

```
int count = 0;
for(int i = 0 ; i< students.length; i++){
//Whenever working with array of objects,
always do a null check
    if(students[i]!= null &&
students[i].isAdult()){
        count++;
    }
}
```

NOTE

Do a **null check** whenever traversing through array of objects to avoid `NullPointerException`.

Using the enhanced loop

```
int count = 0;
for(Student s: students){
    if(s!=null && s.isAdult()){
        count++;
    }
}
```

3. Working With ArrayList of Objects

For the same class Student above.

```
class Student {
    private String name;
    private int age;

    public Student(String name, int age){
        this.name = name;
        this.age = age;
    }

    public int getAge(){
        return age;
    }

    public boolean isAdult(){
        return age > 18;
    }
}
```

The following piece of code creates an `ArrayList` of Student objects and performs operations with it.

CREATING AN ARRAYLIST AND ADDING OBJECTS TO IT

CONCEPT

Code snippet:

```
ArrayList<Student> students = new
ArrayList<Student>();
students.add(new Student("LMN",15));
students.add(new Student("LOP",19));
```

TRAVERSING THE ARRAYLIST USING A NORMAL FOR LOOP

CONCEPT

The code below counts the number of students that are adults.

Code snippet:

```
int count = 0;
for(int i = 0 ; i<students.size(); i++)
{
    if(students.get(i).isAdult()){
        count++;
    }
}
```

NOTE

Common mistakes while traversing an `ArrayList` is to use the `length` instead of `size` and is to do `students[i]` inside the loop to access a Student object.

TRAVERSING THE ARRAYLIST USING THE FOR EACH LOOP

CONCEPT

The for each loop can be used to traverse both arrays and ArrayLists.

The code for counting the number of adults can be seen below.

Code snippet:

```
int count = 0;
for(Student s: students){
    if(s.isAdult()){
        count++;
    }
}
```

4. Wrapper Classes

CONCEPT

Java created wrapper classes for each of the primitive data types. With ArrayLists, you can now work with these wrapper classes if you intend to work with `integers` or `doubles`.

Primitive Type	Wrapper Class
`int`	`Integer`
`double`	`Double`

5. Integer Class Members and Methods

1. `Integer.MIN_VALUE`

The minimum value represented by an int or Integer. Its a **`static` field** of the `Integer` class

Example:
It can be used to find the maximum value in an array.

Part of the code snippet:

```
int max = Integer.MIN_VALUE;
//Rest of the code to find maximum
in a 1D Array
```

2. `Integer.MAX_VALUE`

The maximum value represented by an int or Integer. It's a **`static` field** of the `Integer` class.

Example:
It can be used to find the minimum value in an array.

Part of the code snippet:

```
int max = Integer.MAX_VALUE;
//Rest of the code to find minimum
in a 1D Array
```

3. `Integer(int value)`

This is the **constructor of the `Integer` class** which creates an **Integer object using an `int`.**

Example:

```
Integer obj = new Integer(15);
//obj can be added in a list of
Integer objects
```

4. `public int intValue()`

`intValue` is a **non-static method** of the Integer class which gives the `int` value of the corresponding object.

Example:

```
ArrayList<Integer> list = new
ArrayList<Integer>;
list.add(new Integer(20));
Integer ob = list.get(0);
int val = ob.intValue();
```

6. Double Class Methods

1. `Double(double value)`

This is the **constructor of the `Double` `class`** which creates a **Double object using a double**.

Example:

```
Double obj = new Double(10.2);
//obj can be added in a list of
Double objects
```

2. `public double doubleValue()`

`doubleValue` is a **non-static method** of the Double class which gives the **`double`** value of the corresponding object.

Example:

```
Double v = new Double(10.3);
double m = v.doubleValue();
```

7. Auto Boxing and Auto Unboxing

Java allows to convert from secondary data types to primitive types and the other way round automatically.

Code:

```
Integer val = 20; //Auto boxing
//Implicitly calls the constructor

int value = val; //Auto unboxing
//Implicitly calls the intValue
```

8. Standard ArrayList Algorithms

REMOVAL OF AN ELEMENT FROM AN ARRAYLIST

Whenever an element is removed, the **size of the list changes immediately** and hence it is necessary to reset the counter to not miss out on deleting elements that are next to each other.

The code below removes all the occurrences of the word "Hi" in the ArrayList.

Code:

```
ArrayList<String> list = new
ArrayList<String>();

//Assume some strings are already
added to the list

for(int i = 0 ; i<list.size(); i++)
{
   if(list.get(i).equals("Hi")) {
      list.remove(i);
      i--; //Reset the value of i to
check for consecutive occurrences
   }
}
```

NOTE

In an **incrementing loop** do not forget to perform the operation of **decrementing i** in order to make sure consecutive occurrences in the list are deleted.

Another way to do the same, is by traversing the list from the last value. Note, over here you do not need to do the decrementing.

Code:

```
for(int i = list.size()-1 ; i>=0; i--)
{
   if(list.get(i).equals("Hi")) {
   list.remove(i);
   }
}
```

ADDING AN ELEMENT IN A SORTED ARRAYLIST

If you need to add an element in an already sorted list of objects (they can be sorted on any attribute) then you need to get the correct position by traversing the list and use the `add(index, element)` method to do so.

Remember to make sure you take care of all the possible scenarios.

1. Adding at the beginning of the list
2. Adding within the list
3. Adding at the last position in the list

The code below adds a string in the alphabetical order of already sorted list of Strings.

Code:

```
ArrayList<String> list = new
ArrayList<String>();

//Assume some strings are already
added to the list in sorted form

String tobeAdded = //A String that
needs to be added in the correct
position

for(int i = 0 ; i<list.size(); i++) {

if(tobeAdded.compareTo(list.get(i))<0)
{
   list.add(i, toBeAdded);
//This if condition will take care of
adding the String in the beginning and
in between the values of the list
   return;
 }
}
list.add(toBeAdded);
//For adding at the end if
alphabetically greater than all the
values in the list
```

Table of Contents

1. Introduction

CONCEPT

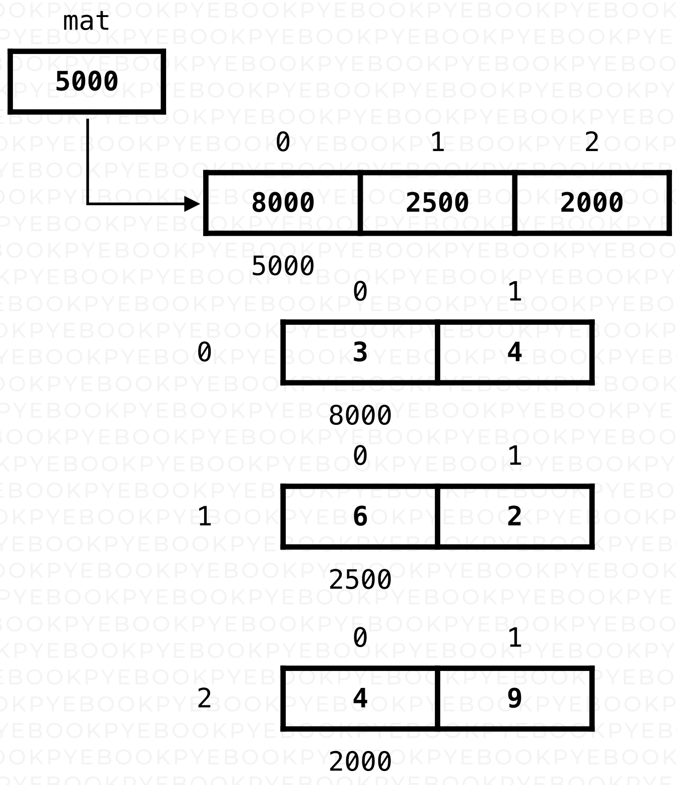
A 2D array (also called a matrix) is an array which has rows and columns. In other words, it is a collection of multiple 1D Arrays.

General declaration of a 2D array:

```
<data type> [ ] [ ] <name of 2D array> ;
or
<data type> <name of 2D array> [ ] [ ];
```

EXAMPLE

```
int [][] mat = {{3,4},{6,2},{4,9}};
```



Notes from the memory diagram:

1. The name of the 2D array is a reference to an array of references.
2. Each reference of the array of references points to a 1D array.
3. So mat is pointing to an array of 3 references and the 0th array has 2 values and so do the 1st and the 2nd arrays.

IMPORTANT 2D ARRAY OPERATIONS

Code:

```
System.out.println(mat.length);
```

Output:

```
3
```

Gives the **number of arrays or rows**

Code:

```
System.out.println(mat[0].length);
```

Output:

```
2
```

Gives the number of values in the 1st array which **is also the number of columns**. Note `mat[1].length` and `mat[2].length` is also 2

Code:

```
System.out.println(mat[1][1]);
```

Output:

```
2
```

Gives the value at array number 1 **(row number)** and value number 1 **(column number)**. Remember numbering starts from 0.

2D ARRAY CREATION USING NEW

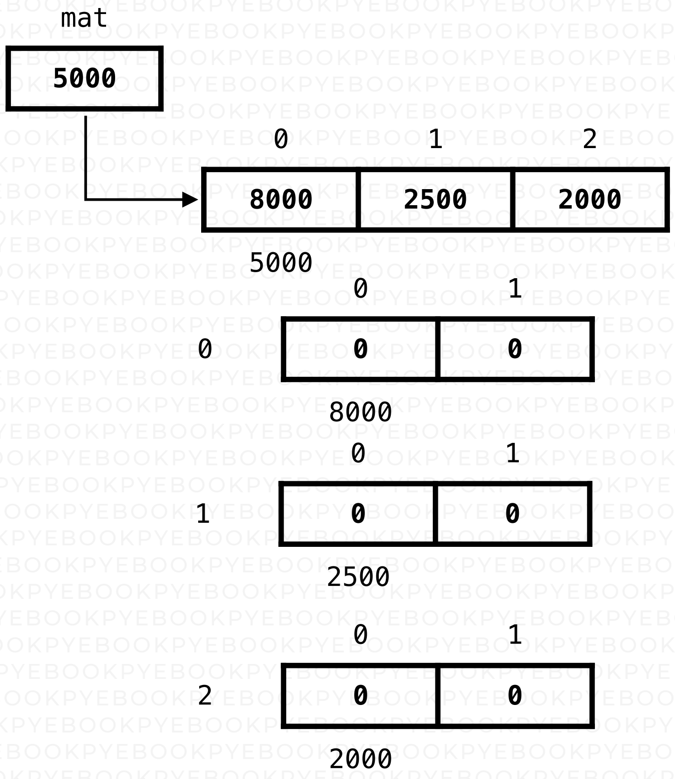
CONCEPT

General declaration of a 2D array

```
<data type> <name of 2D array> [ ] [ ] =
new <data type> [<rows>][<cols>];
```

Creates a 2D array with `<rows>` rows and `<cols>` columns and initialises all values to **`0`** (for int), 0.0 (for double) and false (for boolean).

```
int mat[][] = new int[2][3];
```



2. Traversing Through All the Values of the Matrix

Using a usual for loop

```
for(int i = 0 ; i < mat.length; i++)
   for(int j = 0 ; j < mat[i].length; j++)
      System.out.println(mat[i][j]);
```

Using an enhanced for loop

```
for(int val[]: mat) //val will take each reference from the array mat is pointing to
   for(int v:val) // v will take the value from those references
      System.out.println(v);
```

3. Standard Matrix Algorithms

1. Traversing through all the values in the row **r** .

Code:

```
for(int i = 0; i <mat[r].length; i++)
      System.out.println(mat[r][i]);
```

2. Traversing through all the values in the column **c** .

Code:

```
for(int j = 0; j <mat.length; j++)
      System.out.println(mat[j][c]);
```

3. Print the main diagonal (top left to bottom right of a square matrix (same number of rows and columns).

Code:

```
for(int i = 0; i < mat.length; i++)
      System.out.println(mat[i][i]);
```

4. Print the other diagonal of a square matrix. (top right to bottom left).

Code:

```
for(int i = 0; i <mat.length; i++)
      System.out.println(mat[i]
[mat.length-i-1]);
```

5. Store all the values of a matrix in a 1D array.

Code:

```
int vals[] = new int[mat.length *
mat[0].length];
int pos = 0;
for(int i = 0 ; i < mat.length; I++){
      for(int j = 0 ; j <
mat[i].length; j++) {
            vals[pos] = mat[i][j];
            pos++;
   }
}
```

6. Store the sum of each row in a 1D array.

Code:

```
int vals[] = new int[mat.length];

for(int i = 0 ; i < mat.length; i++){
      for(int j = 0 ; j <
mat[i].length; j++) {
            vals[i] += mat[i][j];
   }
}
```

4. 2D Array of Objects

Work with a 2D array of objects similar to how you would with a 1D array of objects.

Watch out for `null` and **do a null check** to avoid **`NullPointerException`**.

Table of Contents

CONCEPT

Inheritance is the concept of reusing an existing class. In the entire chapter below, the `Student class` is reusing the definition of the `Person class.`

1. Superclasses and Subclasses

CONCEPT & EXAMPLE

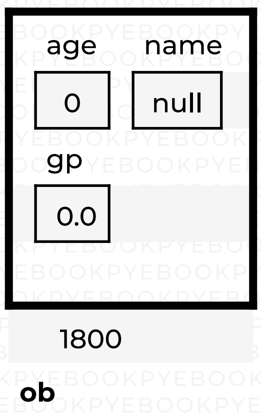
1. The class which is going to be reused is called as the parent class or the super class or the base class.
2. The class which is reusing another class and inheriting the members using the **extends** keyword is called as the child class or the sub class or the derived class.

```
class Person {
    private String name;
    private int age;

    public void display() {
        System.out.println(name +" "+age);
    }
}
class Student extends Person {
    private double gp;

    public void show() {
        System.out.print("Hi");
    }
}

class Main{
    public static void main (String[] args) {
        Student ob = new Student();
        ob.display(); //Will output null 0
        Person p = new Person();
        p.show();
    }
}
```

`Person` is the base class

`Student` is the derived class of Person. It has all the members of Person AND its own members

ob is the reference to the object containing members of all classes



`Person` is the parent class and cannot access members of the derived class. It will give an error

2. Accessing Super Class Members in Subclasses

The classes are modified as below.

```
class Person {

    private String name;

    public String getName() {
        return name;
    }
}

class Student extends Person{

    public void show(){
        System.out.println(name);

        System.out.println(getName());

    }
}
```

getName is also a member of the Student class and since the access modifier is public, you can access like shown

This will result in an error. Note that although name is inherited it cannot be accessed since the access modifier of name is private

3. Method Overriding

CONCEPT & EXAMPLE

1. When base class and derived class non static methods **have same name, same parameters, same return type and same access specifiers then it is said to be method overriding.**

2. When the **derived class instance makes a call to the overridden method, the subclass version is always called.**

```
class Person {
    public void display(){
        System.out.println("Hi");
    }
}
class Student extends Person{

    public void display() {
        System.out.println("Hello");
    }
}

class Main{
    public static void main (String[] args) {
        Student ob = new Student();
        ob.display();
    }
}
```

display is **overridden** here as it has same name, same parameters, same access modifier and same return type

Since ob is the derived class object when we make a call to the overridden method, it will ALWAYS call the derived class version of the method

4. Accessing Base Class Overridden Method in the Subclass Using the Super Keyword

CONCEPT & EXAMPLE

1. In derived class functions, calls can be made to the base class overridden methods by explicitly placing the keyword **super**.

2. Not placing the **super keyword**, will make a call to the overridden method in the sub class itself

`super` makes a call to the overridden method of the base class

```
class Person {

    public void display(){
        System.out.println(“Hi”);
    }
}

class Student extends Person{

    public void display() {
        super.display();
        System.out.println("Hello");
    }
}

class Main{
    public static void main (String[] args) {
        Student ob = new Student();
        ob.display();
    }
}
```

ob calls the derived class' `display`. Inside it the first call is to the super class' `display` followed by printing Hello. So the output will be

Hi
Hello

NOTE

In the case that `class A` is inherited by `class B` and `class B` is inherited by `class C`, then in that case the `super` keyword will only call the method if the immediate super class.

```
class A {
    public void show() { … }
}
class B extends A {
    public void show() {…// super.show() will refer to class A’s show }
}
class C extends B {
   public void show() {..//super. show() will refer to class B’ show and NOT class A’s
show }
}
```

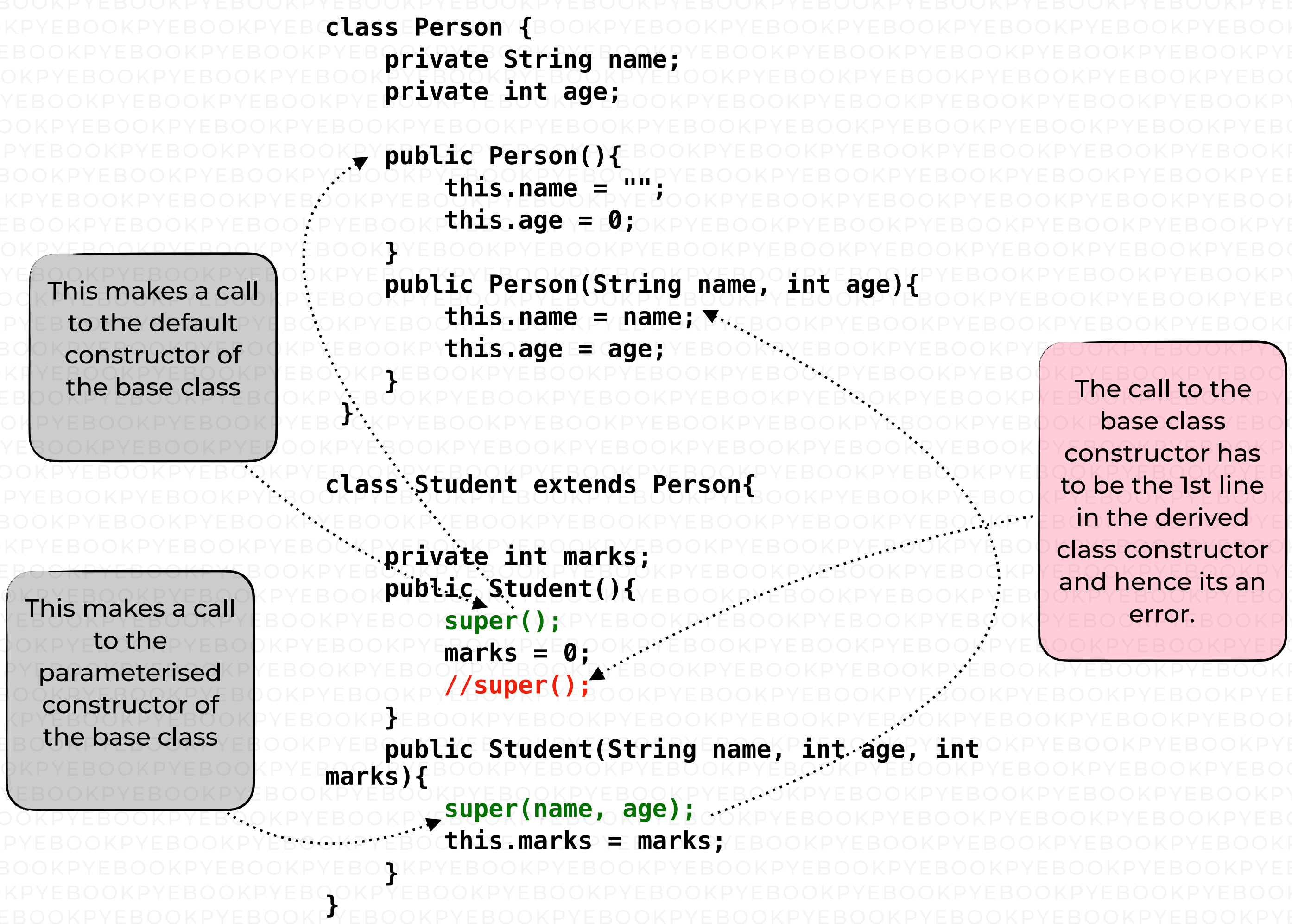
5. Constructors in Inheritance Using Super

CONCEPT & EXAMPLE

1. Whenever an object of the derived class is created, the base class members are created first followed by the derived class members.

2. In the constructors of the derived class, `super` has to be the `first line in the constructor`.

3. `super` makes a call to the constructors of the base class.



```
class Person {
    private String name;
    private int age;

    public Person(){
        this.name = "";
        this.age = 0;
    }
    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}

class Student extends Person{

    private int marks;
    public Student(){
        super();
        marks = 0;
        //super();
    }
    public Student(String name, int age, int
marks){
        super(name, age);
        this.marks = marks;
    }
}
```

This makes a call to the default constructor of the base class
This makes a call to the parameterised constructor of the base class
The call to the base class constructor has to be the 1st line in the derived class constructor and hence its an error.

6. The Object Superclass

1. `Object` is the superclass of ALL the classes in Java.

2. If you do not inherit any class, then your class will inherit the `Object` class by default.

3. All the methods of the `Object` class are available in all the classes that are created.

The following methods of the Object class are important.

1. `public String toString()`

```
class Person extends Object {
    private String name;
    private int age;

    public Person(){
        this.name = "";
        this.age = 0;
    }
}
class Main{
    public static void main (String[] args) {
        Person ob = new Person();

          System.out.println(ob);

    }
}
```

Even if you don't write explicitly, Java does it for you

Since the Person class inherits the Object class and Person class has not overridden the `toString` method, this particular code will perform the operation done b Object's `toString` Which is it prints "<name of theClass>@<hash code code of the reference of the object>"

Printing the reference of a class, implicitly makes a call to the `toString` method as follows `System.out.println(ob.toString());`

WE NOW OVERRIDE THE TOSTRING METHOD IN THE PERSON CLASS

```
class Person extends Object{
    private String name;
    private int age;

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

   public String toString(){
       return name + " "+age;
   }
}

class Main{
    public static void main (String[] args) {
        Person ob = new Person("ABC", 20);
        System.out.println(ob);

    }
}
```

Overriding `toString` method

Since we have overridden the `toString` method, this will call the toString of the Person class now and will output **ABC 20**

2. `public boolean equals(Object o)`

The String class is also a subclass of the Object class and has overridden the equals method which we have used to check whether two Strings are the same or not.

7. Runtime Polymorphism

CONCEPT & EXAMPLE

1. A base class reference can point to the instance of a subclass but not the other way round.
2. Using the base class reference, we can access ,
    1. The base class members.
    2. **ONLY** the overridden methods of the subclass.

```
class Person {
    private String name;
    private int age;

    public Person(){
        this.name = "ABC";
        this.age = 10;
    }

   public void display(){
       System.out.println(“Hi”);
    }
}

class Student extends Person{

    private int marks;
    public Student(){
        super();
        marks = 0;
    }

    public void display() {
        super.display();
        System.out.println("Hello");
    }

    public void show(){
        System.out.println("Subclass");
    }
}

class Main{
    public static void main (String[] args) {
       Person ob = new Student();
       ob.display();
       ob.show();
    }
}
```

Using the reference of the base class, this code will make a call to derived class' overridden method. The output will be
Hi
Hello

Person is the base class and Student is the subclass. Base class reference can point to the object of subclass.

This will give a compiler error. Although ob is pointing to the Student's object it CANNOT access show since its not overridden nor is it the member of Person

8. Another Polymorphism Scenario

Consider the following set of classes.

```
class A {
   public void show() {
     System.out.println("A");
   }
}
class B extends A {
   public void show() {
     System.out.println("B");
   }
}
class C extends A {
  public void show() {
     System.out.println("C");
  }
}

class D extends A {
     public void display() {
       System.out.println("D");
     }
}

class Main {
     public static void main(String arg[]){

     A objects[] = new B[2];

     A obs[] = new A[3];

     obs[0] = new B();
     obs[1] = new C();
     obs[2] = new D();

     obs[0].show();
     obs[1].show();
     obs[2].show();

     obs[2].display();

     B ob2 = new A();

     }
}
```

Since `obs[0]` is pointing to an object of class B and B has overridden show then the **output will be B**

Although `obs[2]` is pointing to the D's object it CANNOT access display since its not overridden nor is it the member of A. This will be an error.

Derived class reference cannot point to a base class instance. This will give an error

objects is an array of A and can store references of objects that belong to class B since B extends A.Hence this line of code is valid.

All these 3 lines of code are valid since B, C and D are derived classes of class A

Since `obs[1]` is pointing to an object of class C and C has overridden show then the **output will be C**

Since `obs[2]` is pointing to an object of class D and D has NOT overridden show it will perform the show as how the base class which is A has done and **hence it will output A**

Table of Contents

1. Introduction

CONCEPT

Recursion is the process of a method calling itself.

Every recursive function has two components:

1. **Base component** - Simplest part of the function which usually returns the answer to the simplest possible input or inputs.

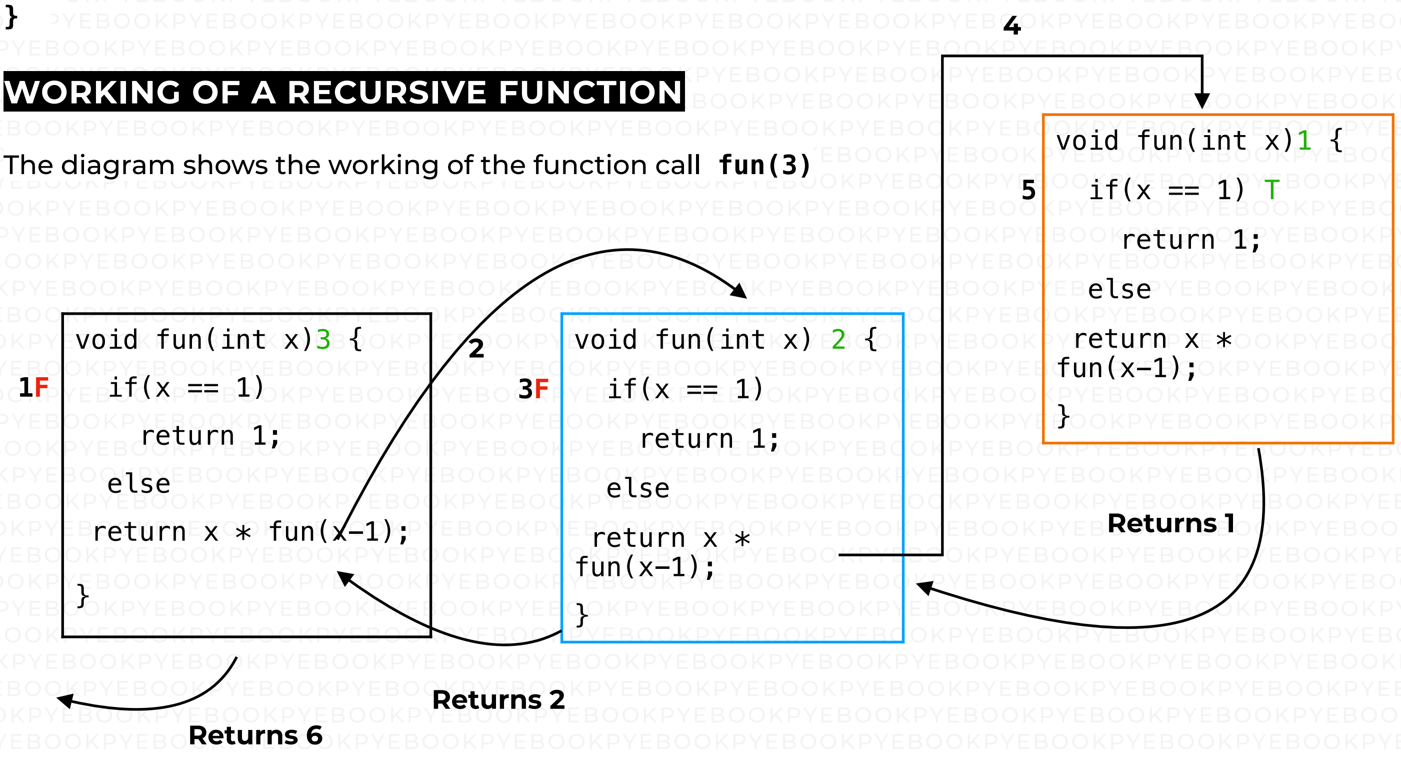
2. **Recursive component** - Breaks down the original problem into a smaller problem of the same type.

RECURSIVE FUNCTION WHICH RETURNS A VALUE

```
public int fun(int x) {
   if( x == 1) {    —> This is called as the base component
      return 1;
   }
   else {
      return x * fun(x-1);  --> This is the recursive component
   }
}
```

WORKING OF A RECURSIVE FUNCTION

The diagram shows the working of the function call **`fun(3)`**



Go through the diagram above along with the explanations below.

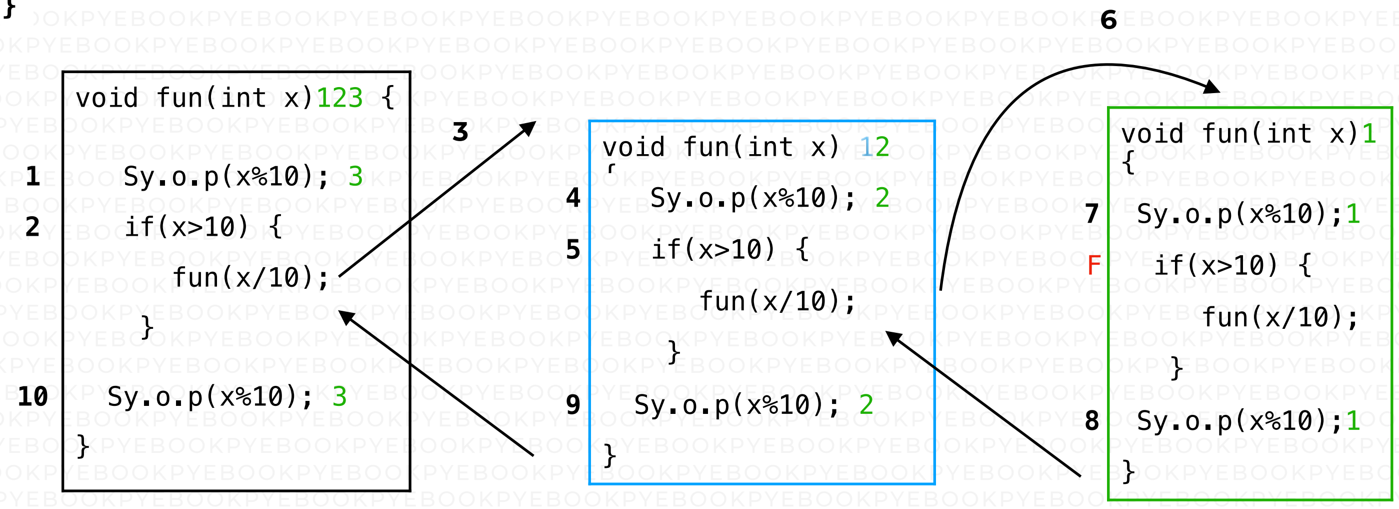
When the function is called the value of x is 3

1. The value of x is checked to be 1. This condition is False.
2. It goes in the else and there is a recursive call, which calls the function again and in that function the x value is 2.
3. The value of x is checked and this time again its False.
4. It goes in the else and another recursive call is made. In that function call, now x is 1.
5. Since the condition is satisfied the last call returns 1.
6. It goes back to the previous call and calculates 2 * 1 and returns it.
7. It goes back to the original call and the value returned now **3 * 2 which is the final answer.**

The function above calculates the **factorial of a number.**

RECURSIVE FUNCTION WITH VOID RETURN TYPE

```
public void fun(int x) {
   System.out.print(x%10);
   if( x > 10) {
      fun(x/10);
   }
   System.out.print(x%10);
}
```



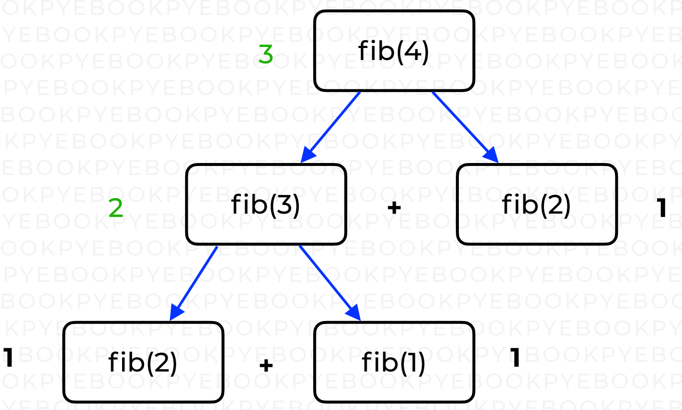
When the function is called the value of x is 123

1. First the `System.out.print` prints 3
2. Since 123 is more than 10,
3. the function is called again and the x value there is now 12.
4. The `System.out.print` prints 2
5. Since 12 is more than 10,
6. The function is called again and the x value there is now 1.
7. The `System.out.print` prints 1
8. Since 1 is not more than 10, it goes to the last `System.out.print` and prints 1 again
9. It goes back to the previous call and prints 2. Remember x value there is still 12
10. It goes back to the original call and prints 3. Remember the x value there is still 123.

Final Output is **321123**

MULTIPLE RECURSIVE CALLS IN THE RECURSIVE COMPONENT

```
public void fib(int x) {
    if(x==1 || x==2) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-1);
    }
}
```



Working of the function for the function call **`fib(4)`**

1. When x is 4 the if condition is not satisfied and then it will return the result of the following recursive component `fib(3) + fib(2)`. First `fib(3)` will be evaluated.
2. When `fib(3)` is getting evaluated x is 3. Since x is neither 1 nor 2, it will return the result to the recursive call `fib(2) + fib(1)`.
3. `fib(2)` is going to return **1.**
4. `fib(1)` is going to return **1**. Once fib(1) is returned, `fib(3)` is now complete with both of its recursive calls and returns the value 2 to the original call.
5. We are back in `fib(4)` now and since `fib(3)` is complete, `fib(2)` is called and it will return **1.**
6. For `fib(4)`, `fib(3)` returned 2 and `fib(2)` returned 1 and now the final answer returned is **3**.

NOTE

1. Tracing back in recursion is key.
2. There might be inputs for which the recursion never stops. In these cases you get a `StackOverflowError`. For example in the function `fib` above, you will get a `StackOverflowError` for value of 0 because the if condition never gets satisfied and the else condition keeps making recursive calls.

2. Binary Search

You can find the algorithm in Unit 6. The recursive code for the same is as shown below

```
int binarySearch(int arr[], int l, int r, int
x)
  {
    if (r >= l) {
    int mid = l + (r - l) / 2;

    // If the element is present at the
    // middle itself
    if (arr[mid] == x)
        return mid;

    // If element is smaller than mid, then
    // it can only be present in left subarray
    if (arr[mid] > x)
        return binarySearch(arr, l, mid - 1, x);

    // Else the element can only be present
    // in right subarray
    return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
present
    // in array
    return -1;
  }
```

3. Merge Sort

The Merge Sort Algorithm below sorts an array of integers into ascending order as follows:

ALGORITHM

mergeSort
This top-level method creates the necessary temporary array and calls the mergeSortHelper recursive helper method.

mergeSortHelper
This recursive helper method uses the Merge Sort Algorithm to sort elements[from] ... elements[to] inclusive into ascending order
1. If there is more than one item in this range,
    a. divide the items into two adjacent parts, and
    b. call mergeSortHelper to recursively sort each part, and
    c. call the merge helper method to merge the two parts into sorted order.
2. Otherwise, exit because these items are sorted.

merge

This helper method merges two adjacent array parts, each of which has been sorted into ascending order, into one array part that is sorted into ascending order:

1. As long as both array parts have at least one item that hasn't been copied, compare the first un-copied item in each part and copy the minimal item to the next position in temp.
2. Copy any remaining items of the first part to temp.
3. Copy any remaining items of the second part to temp.
4. Copy the items from temp[from] ... temp[to] inclusive to the respective locations in elements.

CODE

```
public static void mergeSort(int[] elements) {
    int n = elements.length;
    int[] temp = new int[n];
    mergeSortHelper(elements, 0, n – 1, temp);
}

private static void mergeSortHelper(int[] elements,
int from, int to, int[] temp)
{
        if(from < to){
         int middle = (from + to) / 2;
         mergeSortHelper(elements, from, middle,
temp);
         mergeSortHelper(elements, middle + 1, to,
temp);
         merge(elements, from, middle, to, temp);
        }
}
private static void merge(int[] elements, int from,
int mid, int to, int[] temp)
{
    int i = from; int j = mid + 1; int k = from;
    while (i <= mid && j <= to) {
       if (elements[i] < elements[j]) {
          temp[k] = elements[i];
          i++;
       }
       else {
          temp[k] = elements[j];
          j++;
       }
       k++;
    }
    while (i <= mid) {
       temp[k] = elements[i]; i++;
       k++;
    }
    while (j <= to) {
       temp[k] = elements[j]; j++;
```

```
            k++;
        }
        for (k = from; k <= to; k++) {
            elements[k] = temp[k];
        }
    }
```