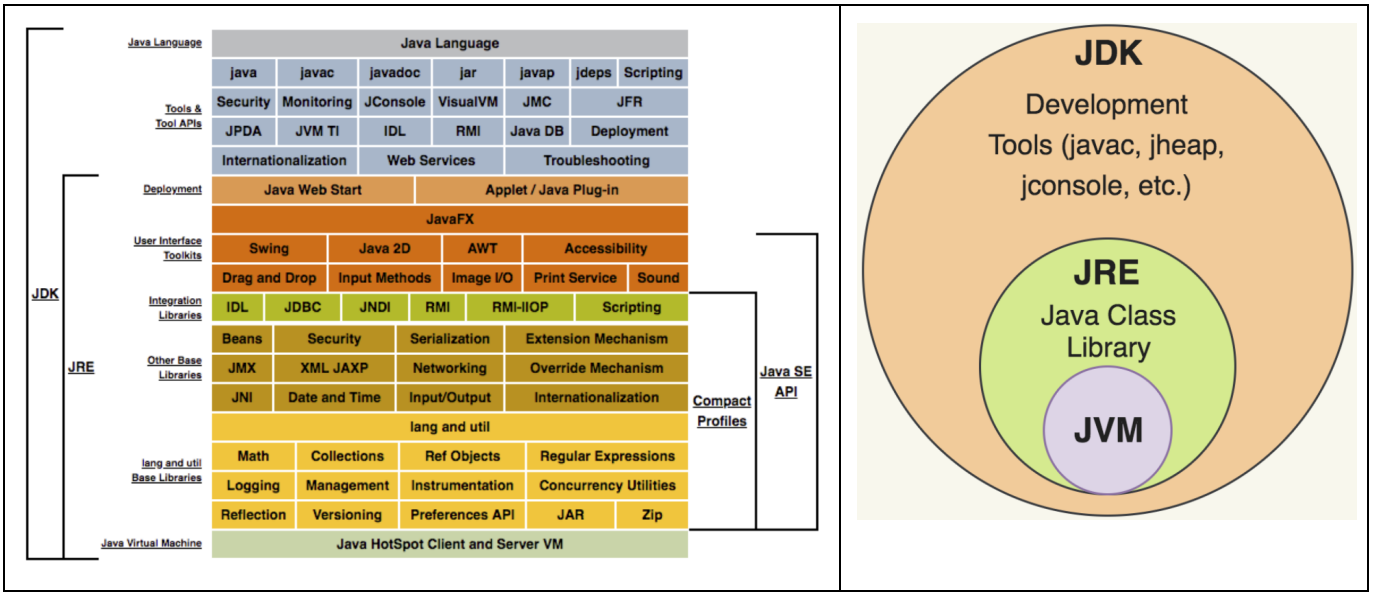
JAVA

JAVA Version History

Version	Release date	End of Public Updates[5]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	January 2019 (commercial) December 2020 (non-commercial)	March 2025
Java SE 9	September 2017	March 2018	N/A
Java SE 10 (18.3)	March 2018	September 2018	N/A
Java SE 11 (18.9 LTS)	September 2018	March 2019 from Oracle Later from OpenJDK	Vendor specific
Java SE 12 (19.3)	March 2019	September 2019	N/A

Legend: Old version | Older version, still supported | **Latest version** | Future release

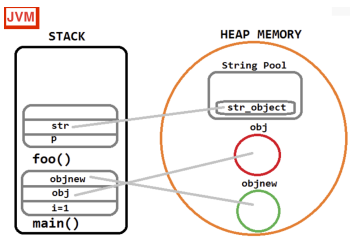
JVM - JRE - JDK



- **Java Virtual Machine (JVM)**

  It is an abstract computing machine that enables a computer to **run** a Java program. JVM performs tasks like loading byte code, code verification, code execution, etc.

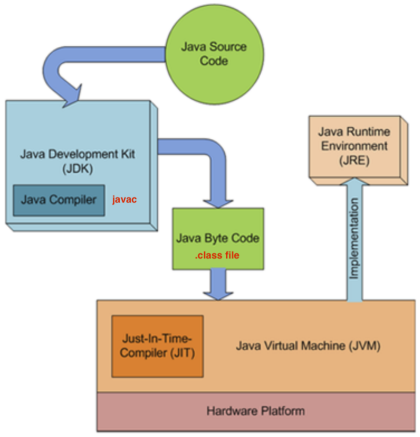
  → JAVA has 2 memory types; Stack and Heap



- **Java Runtime Environment (JRE)**
  - It is a software package (a physical entity) that contains necessary artifacts required to run a Java program. It includes JVM implementation together with an implementation of Java Class Library (rt.jar).
- **Java Development Kit (JDK)**
  - It is a superset of a JRE and contains tools for Java programmers, e.g. javac, jconsole, jheap, jps, jvisualvm...

JDBC → library like pg-promise (to connect with database)

How Java Works?



We have HelloWorld.java file, we should compile it

macs-MBP:desktop mac$ **javac HelloWorld.java //compile**

It creates **HelloWorld.class** on desktop

Then;

macs-MBP:desktop mac$ **java HelloWorld //run the code**

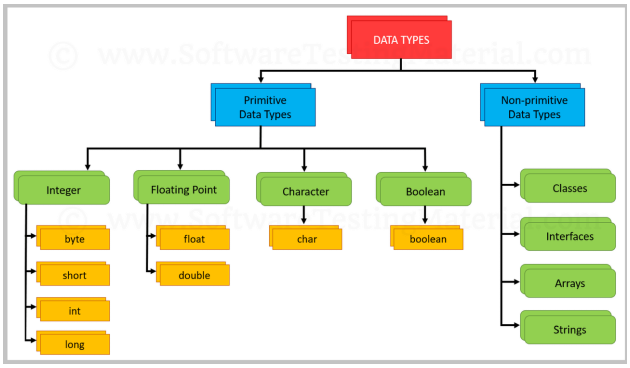
```
Hello World!
My name is Mike
How are you? I am OK
Then Bye
```

My first java program

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //ln --> new line
        System.out.println("My name is Mike");
        System.out.print("How are you?"); //-->continue to previous line
        System.out.println(" I am OK");
        System.out.println("Then Bye");
    }
}
```

```
Hello World!
My name is Mike
How are you? I am OK
Then Bye
```

DATA TYPES



Primitive		Range of values	Default Value
byte	1 byte	-128 to 127	0
short	2 byte	-32768 to 32767	0
int	4 byte	-2,147,483,648 to -2,147,483,648	0
long	8 byte	-9,223,372,036,854,775,808 to ….	0
float	4 byte	3.4e - 038 to 3.4e+038	0.0f
double	8 byte	1.7e - 308 to 1.7e+038	0.0d
char	2 byte	True or false	false
boolean	1 bit		\u0000

(Non-Primitive → Reference)

```java
public class DataTypes {
    public static void main(String[] args) {
        //byte
        byte b1 = 25;
        byte b2 = -128;
        byte b3 = 127;
        System.out.println(b1 + "|" + b2 + "|" + b3);
        //byte b4 = 128; --> error, byte works up to 128

        //short
        short s1 = 500;
        short s2 = -32768;
        short s3 = 32767;
        //short s4 = 32768; Error
        System.out.println(s1 + "|" + s2 + "|" + s3);

        //int
        int i1 = -50000;
        int i2 = 100000;
        int i3 = 100;
        System.out.println(i1 + "|" + i2 + "|" + i3);

        //long
        long l1 = 100_000_000;
        long l2 = 214567898765431L; // or 214567898765431
        System.out.println(l1 + "|" + l2);

    //float: floating points types have double as default type
        float f1 = 5.2f;
        double d1 = 5.2;
        System.out.println(f1 + "|" + d1);

        //boolean
        boolean bool = true;
        boolean bool2 = false;
        //char
        char c1 = 'D';
        char c2 = '2';
        char c3 = '@';
        char c4 = '\u00A9'; //https://unicode-table.com/en/
        System.out.println(c4); //©
    }
}
```

Declaring and Initializing Variables

```
        int numberOfAnimals;
        numberOfAnimals = 100;
or
        int numberOfAnimals = 100;
```

```
        String s1,s2;
        String s3 = "yes",s4="no";

        int i1, i2, i3 = 0;
       //we didn't initialize i1,i2 but i3 is declared and initialized

        int num, String value; //Does not compile
```

Identifiers

There are 3 rules to remember for legal identifiers:

1. The name must begin with a letter or the symbol **$** or **_**
2. Subsequent characters may also be numbers
3. You can **not** use the same name as a Java **reserved** word.

Illegal Identifiers

```
3DPointClass
hollywood@Vine
*$coffee
public    → Public is ok (not reserved)
```

OPERATORS IN JAVA

1. Arithmetic Operators

NAME	OPERATOR	PURPOSE
Addition	+	Adds Value
Subtraction	-	Subtract Value
Division	/	Divides
Multiplication	*	Multiplies
Modulus	%	Divides and remains

```
public class ArithmeticOperators {
    public static void main(String[] args) {
        int i1 = 12;
        int i2 = 5;
        float f1 = 15.5f;
        double d1 = 20.5;
        String msg = "Hello";

        int res1 = i1+i2;
        float res2 = i1 + f1;
        double res3 = i2 + d1;
        String res4 = msg + f1;

        System.out.println(res1); //17
        System.out.println(res2); //27.5
        System.out.println(res3); //25.5
        System.out.println(res4); //Hello15.5
```

CASTING

- Casting is the temporary conversion of a variable from its original data type to some other data type.
- With primitive data types if a cast is necessary from a less inclusive data type to a more inclusive data type it is done automatically.
- If a cast is necessary from a more inclusive to a less inclusive data type the class, it must be done explicitly by the programmer.


```
int number;
double dval= 32.33;
number= (int)dval;
```

Type in which you want to convert

Variable name Which you want to convert

```
        System.out.println("--------------------");

        byte b1 = 12; //implicit casting is done by Java
        int i3 = 12;
        byte b2 = 20;
        // the result of an expression involving anything
int-sized or smaller variable is always an int. If you add two
bytes together, you will get an int and in case of arithmetic
addition, multiplication, subtraction, and division of
integral variables (byte or short) compile process the value
as int, so you need to cast it.

        byte resByte = (byte)(b1+b2);
        byte resByte1 = 20+12;
        byte res7 = 126+1;
        byte res8 = (byte)(126+19);
        System.out.println(res8);   //-111
    }
}
```

2. Unary Operators

NAME	OPERATOR	PURPOSE
Unary Plus	+	Indicates positive value
Unary Minus	-	Negates an expression or value
Increment	++	increments a value by 1
Decrement	--	Decrements a value by 1
Logical Complement	!	Inverts the value of a boolean

```
public class UnaryOperators {
    public static void main(String[] args) {
        int i1 = +10;
        int i2 = 10;
        int i3 = -100;
        int i4 = -(i1+i2);
        System.out.println(i1+"|"+i2+"|"+i3+"|"+i4);
        //10|10|-100|-20

        //increment operator
        int i = 10;
        int j = 5;
        i++;
        --j;
        System.out.println(i+"|"+j);  //11|4

        int res = i++ + --j;
        System.out.println(res + "|" + i + "|" + j); //14|12|3
        boolean flag = true;
        System.out.println(!flag);  //false
        System.out.println(!!flag); //true
    }
}
```

3. Relational Operators

OPERATOR	NAME
==	Equality
!=	Inequality
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal

```
public class RelationalOperators {

    public static void main(String[] args) {

        int i1 = 5;
        int i2 = 10;

        System.out.println(i1==i2); //false
        System.out.println(i1!=i2); //true
        System.out.println(i1>=i2); //false
        System.out.println(i1<=i2); //true

    }
}
```

4. Logical Operators

OPERATOR	NAME	TRUTH TABLE	
&&	AND		• && operator supports short-circuit evaluations but & operator does not. • IF the first operand to && operator evaluates to false, the result can never be true, hence && does not evaluate the second operand. • But & operator evaluates both the operands before returning an answer.
&	Bitwise AND		
||	OR		• || operator supports short-circuit evaluations but | does not • If the first operand to || operator evaluates to true, the result can never be false, hence || does not evaluate the second operand. • But | operator evaluates both the operands before returning an answer
|	Bitwise Inclusive OR		
^	Bitwise Exclusive OR		• The result of a bitwise logical exclusive OR operation is true, if and only if one operand is true and the other is false • Note that both operands must always be evaluated in order to calculate the result of a bitwise logical exclusive OR.

AND / Bitwise AND

	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

OR / Bitwise Inclusive OR

	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Bitwise Exclusive OR

	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	FALSE

5. Assignment Operators

OPERATOR	Expression	MEANING
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

```java
public class AssignmentOperators {

    public static void main(String[] args) {

        int i1 = 100;
        i1+=1;
        System.out.println(i1); //101
        i1-=2;
        System.out.println(i1); //99
        i1*=3;
        System.out.println(i1); //297
        i1/=4;
        System.out.println(i1); //74

    }
}
```

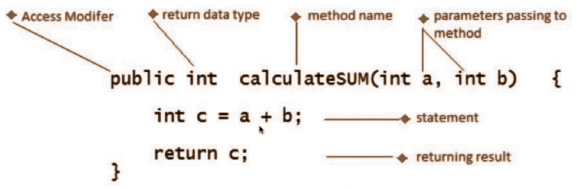
- **Operator Precedence in Java**
  - !, +, - → First (unary operators)
  - *, /, % → Second
  - +, - → Third
  - <, <=, >=, > → Fourth
  - ==, != → Fifth
  - && → Sixth
  - || → Seventh
  - = → Lowest (assignment operators)

CONTROL FLOW STATEMENTS

1. Selection Statements:
   a. if, if - else, if-else if - else
   b. switch case
2. Looping Statements:
   a. for
   b. while, do-while
3. Branching Statements:
   a. break
   b. continue

METHODS

- A method is a collection of statements that perform some specific task and return result to the caller.
  (JS function → We can reuse the function rather than repeating the same set of statements.)




```java
public class CalculateSum {

    public static void main(String[] args) {
        //calling our methods
        calculateSum(4,5);
        System.out.println(calculateSum2(5,10));

    }
    //void --> it does not return anything
    public static void calculateSum(int a, int b) {
        int sum = a+b;
        System.out.println(sum);
    }
    public static int calculateSum2(int x, int y) {
        int sum = x+y;
        return sum;
    }
}
```

METHOD OVERLOADING

■ It is a feature that allows us to have more than one method with the **same name**, so long as we use **different parameters**,
■ It improves the code **readability** and **re-usability**.
■ It is **easier to remember** one method name.
■ Overloaded methods give programmers the **flexibility** to call similar method with different types of data.

```
public class CalculateScore {
    public static void main(String[] args) {
        int newScore = calculateScore("Mike", 500);
        System.out.println("New score is " + newScore);
        calculateScore(75);
        calculateScore();
    }
    public static int calculateScore(String playerName, int score) {
        System.out.println("Player "+playerName+" scored "+score+" points");
        return score * 100;

    }
    public static int calculateScore(int score) {
        System.out.println("Unnamed Player Scored " + score + " points");
        return score*100;
    }
    public static void calculateScore() {
        System.out.println("No Player Name, No Player Score");
    }
}
```

SWITCH CASE

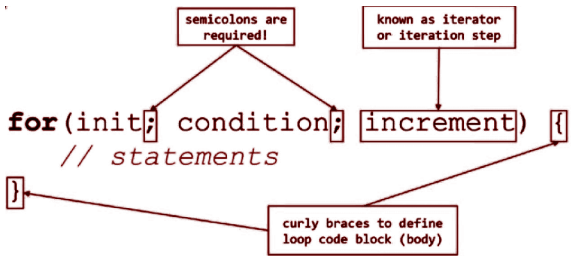
It is used to compare the value of a variable with multiple values and execute some statements based on the ***match***.

```
switch (expression) {
        case constant1:
            title = "Level 1";
            statement 2;
            break;
        case constant2:
            statement 1;
            statement 2;
            ...
            break;

        default:
            statement 1;
            statement 2;
            break;
}
```

```
public class SwitchClass {

    public static void main(String[] args) {

        int switchValue=3;
        switch(switchValue) {
        case 1:
            System.out.println("value was 1");
            break;
        case 2:
            System.out.println("value was 2");
            break;
        case 3: case 4: case 5:
            System.out.println("value was 3, or 4 , or 5");
            break;
        default:
            System.out.println("was not correct value");
            break;
        }
    }
}
```

FOR LOOP

- I am using loop in my project while checking our database. (Data Driven Framework)




```
public class CalculateInterest {

    public static void main(String[] args) {
        //How much is the interest for $10000 with %2, %3, %4, %5 rate
        for(int i=2;i<=5;i++) {
         System.out.println("Interest is " + calculateInterest(10000, i));
        }
    }
    public static double calculateInterest(double amount,
double rate) {
        return amount * (rate/100);
    }
}
```

WHILE, DO WHILE LOOP, BREAK, CONTINUE

- **While loop** checks the condition at the start before executing the block.

```
while (condition) {
    // statements
}
```
Curly braces to define loop code block (body)

- With the **do while loop** the code block is *executed at once* and then the condition is checked.

```
do {
    // statements
} while (condition);
```
Semicolon is required!

Curly braces to define loop code block (body)

- With the **continue** keyword the loop will bypass the part of code block that is below the continue keyword and continue with the next iteration.
- With the **break** keyword we can exit the loop depending on the condition that we are checking.

TASK:
Write a method with the name sumDigitsthat has one int parameter called number.
▪If parameter is >= 10 then the method should process the number and return sum of all digits, otherwise return -1 to indicate an invalid value.
▪The numbers from 0-9 have 1 digit so we don't want to process them, also we don't want to process negative numbers, so also return -1 for negative numbers.▪Sample Outputs▪sumDigits(125) should return 8 since 1 + 2 + 5 = 8.
▪sumDigits(1) should return -1 as per requirements described above.
▪Hint: Use n % 10 to extract the least-significant digit.Use n = n / 10 to discard the least-significant digit.

```
public class Task {
    public static void main(String[] args) {
        System.out.println(sumDigits(125)); //8
        System.out.println(sumDigits(-125)); //-1
        System.out.println(sumDigits(32125)); //13
    }
    public static int sumDigits(int number) {
        if(number<10) {
            return -1;
        }
        int sum=0;
        while(number>0) {
            int digit=number%10;//125%10 = 5 //12%10=2   //1%10=1
            sum=sum+digit;      //0+5=5      //5+2=7     //7+1=8
            number=number/10;   //125/10=12  //12/10=1   //1/10=0
        }
        return sum;
    }
}
```

PARSING VALUES FROM A STRING

```
String numberAsString = "2018";
System.out.println(numberAsString); //2018

int number = Integer.parseInt(numberAsString);
System.out.println(number);

numberAsString += 1; //20181
number += 1; //2019

double number = Double.parseDouble(numberAsString);
System.out.println(number); //2019.0
```

Integer is a class and parseInt is a method
Wrapper class Integer contains some useful static methods like the one that we're using parse int now this method parseint will try to convert the string that we're passing as an argument it's a number of string here into an integer.

READING USER INPUT

```
//1.Create , scanner using the InputStream available
Scanner scanner = new Scanner( System.in);

//2.Don't forget to nromnt the user
System.out.print( "Type some data for the program: ");

//3.to read a line of text from the user.
String input = scanner.nextLine();

//4.Now, you can do anything with the input string that
you need to.
//Like, output it to the user
System.out.println("input = " + input);
```

SPECIAL CONDITION →

```
import java.util.Scanner;  //import java.util.*;

public class UserInput {
    public static void main(String[] args) {
        //create our scanner object
        Scanner scanner = new Scanner(System.in);
        //ask the user
        System.out.println("Enter a byte value:  ");
        byte b1 = scanner.nextByte();

        System.out.println("Enter a short value:  ");
        short s1 = scanner.nextShort();

        System.out.println("Enter a boolean value:  ");
        boolean bool = scanner.nextBoolean();

        System.out.println(b1+"|"+s1+"|"+bool);
        //for example: 5|4|true
        scanner.close();
    }
}
```

```
import java.util.Scanner;   //import java.util.*;

public class UserStringInput {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a int value:  ");
        int i1 = scanner.nextInt();

        //if we dont use nextLine() after int, STRING does not work
        scanner.nextLine();

        System.out.println("Enter your name: ");
        String str = scanner.nextLine();
        System.out.println(str);
        scanner.close();
    }
}
```

TASK:
• Read 10 numbers from the console entered by the user and print the sum of those numbers.
• Create a Scanner
• Use the hasNextInt() method from the scanner to check if the user has entered an int value.
• If hasNextInt() returns false, print the message Invalid Number. Continue reading until you have read 10 numbers.
• Use the nextInt() method to get the number and add it to the sum.
• Before the user enters each number, print the message "Enter number #x:" where x represents the count,
i.e. 1, 2, 3, 4, etc.
• For example, the first message printed to the user would be "Enter number #1:", the next "Enter number #2: ", and so on.
Hint:
• Use a while loop.
• Use a counter variable for counting valid numbers.
• Close the scanner after you don't need it anymore.

```java
import java.util.Scanner;   //import java.util.*;

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

        Scanner scanner = new Scanner(System.in);
        int sum=0;
        int counter=1;

        while(counter<11) {
            System.out.println("Enter your number-"+counter);
            boolean bool = scanner.hasNextInt();
            if(bool) {
                counter++;
                int number=scanner.nextInt();
                sum=sum+number;
            }else {
                System.out.println("Invalid Number");
            }
            scanner.nextLine();
        }
        System.out.println("Your total is: " + sum);
        scanner.close();
    }
}
```

ARRAYS

- An array is a data structure that allows you to store multiple values of the same type into a single variable.
- We use one variable to store multiple items of same/compatible type and we use the same variable to modify the items.

DECLARE

1. Specify the type(primitive or object), and then specify square brackets[ ] and finally the array variable name.

```java
int[] scores;
String[] names;
Student[] students;
```

2. Specify the type(primitive or object), and then specify the array variable name which is followed by square brackets [ ]

```java
int scores[];
String names[];
Student students[];
```

INSTANTIATE

To instantiate one dimensional array, use the keyword **new**, followed by type and then finally specify the size within square brackets [ ]

1. Declare and instantiate in separate statements

```java
int[] scores;
scores = new Int[4];
```

```java
String[] names;
names = new String[5];
```

2. Declare and instantiate in single statement.

```java
int[] scores = new int[4];
String[] names = new String[5];
```

→ It is illegal to include the size of the array in the declarations, this statement will give compilation error → `int[5] scores;`

INSTANTIATION WITH DATA

```java
int[] scores = new int[] {85,70,95,90};
OR
int[] scores = {85,70,95,90};
```

In first version, never specify the size.

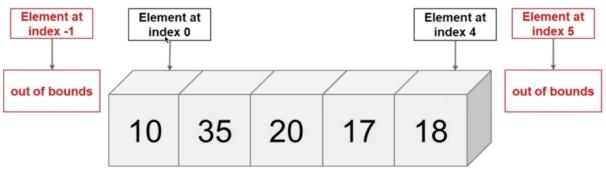
```java
int[] scores = new int[4]{85,70,95,90}; //wrong way
```

ASSIGN VALUES TO ARRAYS

```java
int[] scores; //Declare an int array
scores = new int[3]; //Instantiate an int array of 3 elements
scores[0] = 85; //Assigns 85 to 1st element
scores[1] = 70; //Assigns 70 to 2nd element
scores[2] = 95; //Assigns 95 to 3rd element

System.out.println(scores[0]); //Prints 1st array element
System.out.println(scores[1]); //Prints 2nd array element
System.out.println(scores[2]); //Prints 3rd array element
```

```java
int[] myArray = {10, 35, 20, 17, 18};
myArray[5] = 55;    // out of bounds
```

- Accessing index out of range will cause error in other words **ArrayIndexOutOfBoundsException**
- We have **5 elements** and **index range is 0 to 4**



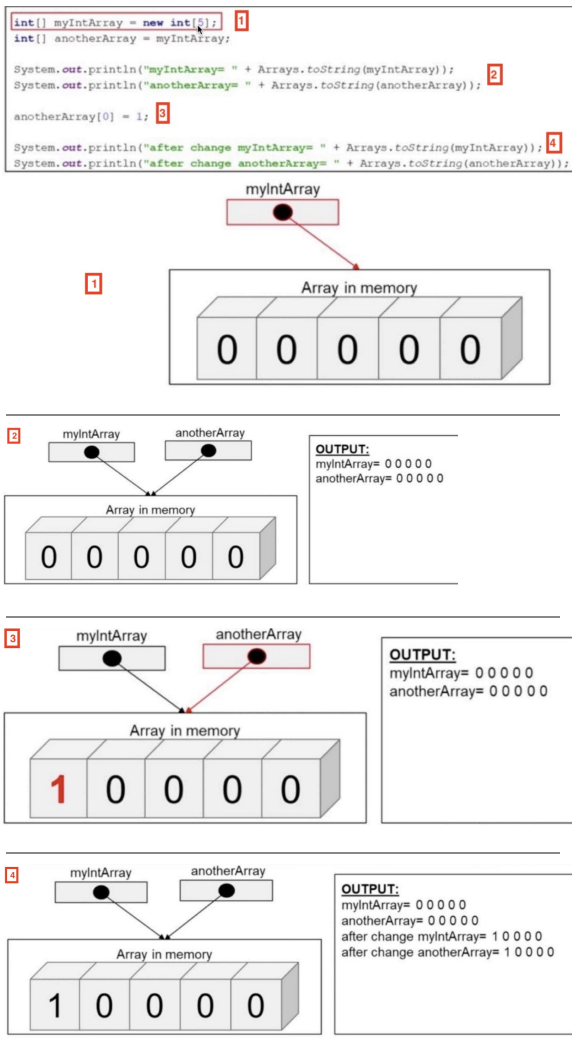
LENGTH PROPERTY OF AN ARRAY

```java
int[] scores = new int[4];
System.out.println(scores.length); //prints 4
```

Relationship between last index and length of the array.

last index == length of the array -1

REFERENCE TYPES vs VALUE TYPES




```
import java.util.Arrays;

public class ReferenceClass {

    public static void main(String[] args) {

        int myIntvalue=10;
        int anotherIntValue=myIntvalue;

        System.out.println("myIntValue:" + myIntvalue);
        System.out.println("anotherIntValue:" + anotherIntValue);

        anotherIntValue=anotherIntValue+1;

        System.out.println("myIntValue:" + myIntvalue);
        System.out.println("anotherIntValue:" + anotherIntValue);

        System.out.println("------------");

        int[] myIntArray=new int[5];
        int[] anotherArray=myIntArray;

        System.out.println("myIntArray:" +
Arrays.toString(myIntArray));
        System.out.println("anotherArray:" +
Arrays.toString(anotherArray));

        anotherArray[0]=1; //we changed the reference array

        System.out.println("myIntArray:" +
Arrays.toString(myIntArray));
        System.out.println("anotherArray:" +
Arrays.toString(anotherArray));
    }
}
```

```
myIntValue:10
anotherIntValue:10
myIntValue:10
anotherIntValue:11
------------
myIntArray:[0, 0, 0, 0, 0]
anotherArray:[0, 0, 0, 0, 0]
myIntArray:[1, 0, 0, 0, 0]
anotherArray:[1, 0, 0, 0, 0]
```

FOR EACH LOOP

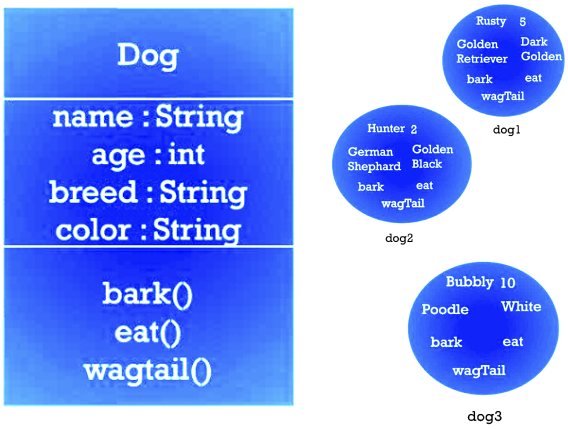
```
for(data_type variable : array | collection){}
```

```
public class forEachClass {
    public static void main(String[] args) {
       int[] marks= {125,132,95,116,110};
       int highest_mark=maximum(marks);
       System.out.println(highest_mark);
   }
   public static int maximum(int[] array) {
       int max=array[0];
       for (int i=0; i<array.length; i++) {}  // for iterator loop

       for(int num:array) {    // for each loop
           if(num>max) {
               max=num;
           }
       }
       return max;
   }
}
```

OBJECT ORIENTED PROGRAMMING (Objects are interacting with each other; OOP → for reuse)

- **CLASSES**

A class is a blueprint or template of the objects of same type




```java
//Dog.java file --> Declaring the class--> 1st STEP
public class Dog {
   public String name;
   public int age;
   public String breed;
   public String color;

   public void bark(){
       System.out.println(name+" barking");
   }
   public void eat(){
       System.out.println(name+ " eating ");
   }
   public void wagTail(){
       System.out.println(name+" is wagging tail");
   }
}
```

```java
//Main.java file --> METHOD --> 2nd STEP
public class Main {
    public static void main(String[] args) {

    Dog dog1 = new Dog();
    dog1.name = "Bubbly";
    dog1.age = 5;
    dog1.breed = "Poodle";
    dog1.color = "White";
    System.out.println(dog1.name + ":" + dog1.age + ":" +
dog1.breed + ":" + dog1.color);

    dog1.bark();
    dog1.eat();
    dog1.wagTail();
    }
}
```

- **CONSTRUCTOR**

- Objects are also known as instances
- To create an instance of the class, new keyword is used.
  For example, to create two instances of Dog class, following statements are used.
  Dog d1 = new Dog();
  Dog d2 = new Dog();
- **new** keyword invokes the constructor of the class
- Java compiler provides a constructor, if no constructor have been defined for the class. This is known as **default constructor.**
- **No return type**

If we put this() in any constructor → it should check without parameter constructor and run it!
this. → go to constructor

If our object property is never changed; we use static word;
public static String *university* = "Cybertek";
I assigned university on my class, so whenever I created new object, university automatically belongs to object.
Constructor is used to set object properties not class (static) properties.

```java
//Dog2.java file --> Declaring the class
public class Dog2 {
   public String name;
   public int age;
   public String breed;
   public String color;

   public Dog2() {  // --> default constructor --with no parameters
       this("x",15,"y","t");
       name= "Hunter";
       age=15;
       breed="bull";
       color="brown";
   }
   public Dog2(String name, int age, String breed, String color) {
       System.out.println("I am running contructor");
       //this();
       this.name = name;
       this.age = age;
       this.breed = breed;
       this.color = color;
   }
}
```

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

    Dog2 dog = new Dog2(); //It goes to default constructor&run it
    System.out.println(dog.name + ":" + dog.age + ":" + dog.breed + ":" + dog.color);

    Dog2 dog2 = new Dog2();
    System.out.println(dog2.name + ":" + dog2.age + ":" + dog2.breed + ":" + dog2.color);

    Dog2 dog3 = new Dog2("Rusty", 20, "Bulldog", "Black");
    System.out.println(dog3.name + ":" + dog3.age + ":" + dog3.breed + ":" + dog3.color);
    }
}
```

```
I am running contructor
Hunter:15:bull:brown
I am running contructor
Hunter:15:bull:brown
I am running contructor
Rusty:20:Bulldog:Black
```

CONSTRUCTOR GOOD EXAMPLE

```
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    // 1st constructor
    public Rectangle() {
        this(0, 0); // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int width, int height) {
        this(0, 0, width, height); // calls 3rd constructor
    }

    // 3rd constructor
    public Rectangle(int x, int y, int width, int height) {
        // initialize variables
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

We have 3 constructors.

- The 1st constructor calls the 2nd, the 2nd constructor call the 3rd constructor, and the 3rd constructor initializes the instance variables.
- No matter what constructor we call, the variables will always be initialized in 3rd constructor.
- This is known as constructor chaining, the last constructor has the "responsibility" to initialize the variables.

BAD EXAMPLE

All three constructors initialize variables and there is repeated code in every constructor. We are initializing variables in each constructor with some default values.

```
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
    }

    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

ENCAPSULATION

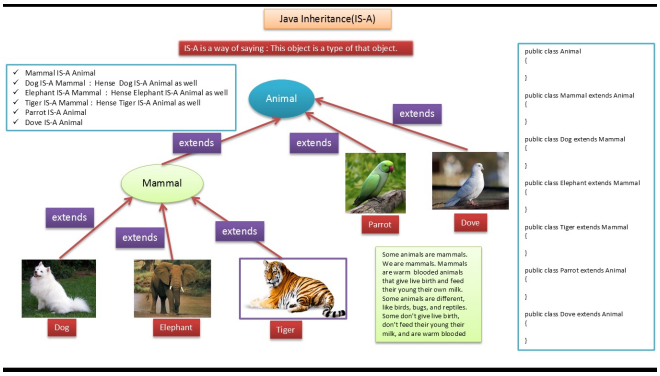
- **It is all about wrapping variables and methods in one single unit. Data hiding.**
  1. **Declare variables of a class as private**
  2. **Provide public getter and setter methods to modify and view the variables values.**

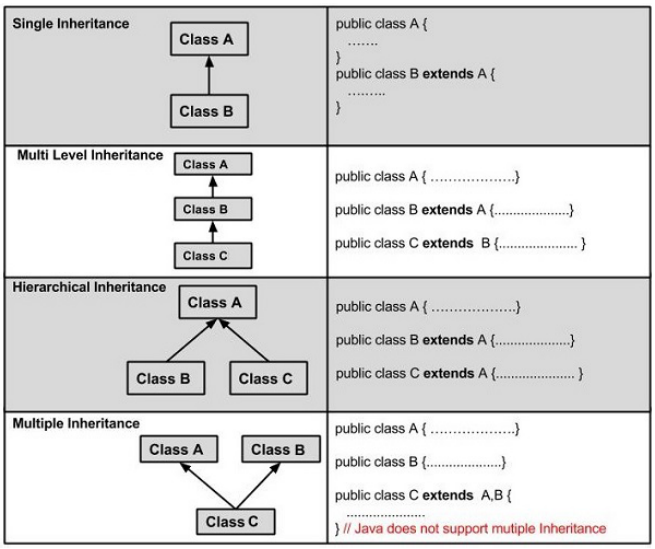
```
public class EncapsulationClass {

    public static void main(String[] args) {

        Car porsche = new Car(); //we invoke the construct

        porsche.setModel("911"); //I set it

        String model = porsche.getModel();

        System.out.println(model);

    }
}
```

```
public class Car {
    private int doors;
    private int wheels;
    private String model;
    private String engine;
    private String color;

    public int getDoors() {
        return doors;
    }
    public void setDoors(int doors) {
        this.doors = doors;
    }
    public int getWheels() {
        return wheels;
    }
    public void setWheels(int wheels) {
        this.wheels = wheels;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        if (model.equals("911")) { //.equals for string (JS=> `==`)
            this.model = model;
        }else {
            this.model = "invalid";
        }
    }
//if we don't use getter and setter method for engine and color, we can not access to view or change them
}
```

INHERITANCE

Why we need? → If we have common things, I don't need to write everything again, so we can use it by inherited






```java
public class Animal {

    private String name;
    private int brain;
    private int body;
    private int size;
    private int weight;

    public void eat() {
        System.out.println("Animal.eat() called");
    }
    public void move(int speed) {
        System.out.println("Animal.move() called. Animal is moving
at " + speed);
    }

    //we don't use setter, because I set them in the constructor
    public Animal(String name, int brain, int body, int size, int
weight) {
        this.name = name;
        this.brain = brain;
        this.body = body;
        this.size = size;
        this.weight = weight;
    }
    //getter methods
    public String getName() {
        return name;
    }
    public int getBrain() {
        return brain;
    }
    public int getBody() {
        return body;
    }
    public int getSize() {
        return size;
    }
    public int getWeight() {
        return weight;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog("Yorkie",1,1,8,20,2, 4, 1, 20, "Silky");
        dog1.eat();
        dog1.walk();
        dog1.run();
    }
}
```

```
Animal.eat() called
Dog.chew() called
Dog.walk() called
Dog.run() called
Animal.move() called. Animal is moving at 50
```

```java
public class Dog extends Animal{

    private int eyes;
    private int legs;
    private int tail;
    private int teeth;
    private String coat;

    public Dog(String name, int brain, int body, int size, int
weight, int eyes, int legs, int tail, int teeth,
            String coat) {
        super(name, brain, body, size, weight); //these are already
assigned in the animal class
        this.eyes = eyes;
        this.legs = legs;
        this.tail = tail;
        this.teeth = teeth;
        this.coat = coat;
    }

    public void chew() {
        System.out.println("Dog.chew() called");
    }
    public void walk() {
        System.out.println("Dog.walk() called");
    }
    @Override //Annotations
    public void eat() {
        System.out.println("Dog.eat() called");
        super.eat();     //super. --> go to parent class (this.--> go
to constructor). If we don't use super and if we have eat() method
for Dog also, then eat() called both parent and child one.
        chew();
    }
    public void run() {
        System.out.println("Dog.run() called");
        super.move(50);
    }
}
```

IMPORTANT NOTE:

```java
public Dog(String name, int size, int weight, int eyes, int legs, int tail, int teeth, String coat) {
    super(name, 1, 1, size, weight);
    this.eyes = eyes;
    this.legs = legs;
    this.tail = tail;
    this.teeth = teeth;
    this.coat = coat;
}
```

We can use also **"1"** and we don't need to initialize **int brain and int body**→ because the dog has already brain and body.

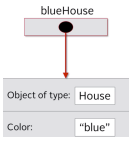
REFERENCE vs OBJECT vs INSTANCE vs CLASS

- A **class** is basically a blueprint for a house, using the blueprint (plans) we can build as many houses as we like based on those plans.
- Each house you build (in other words **instantiate** using the new operator) is an **object** also known as **instance**.
- Each house you build has an address (a physical location). In other words if you want to tell someone where you live, you give them your address. This is known as a **reference**.
- You can copy that **reference** as many times as you like but there is still just one house.
- we can pass **references** as **parameters** to **constructors and methods.**

```
public class main {
    public static void main(String[] args) {
        House blueHouse = new House( "blue" );          1-->

        House anotherHouse = blueHouse;                 2-->

        // Two references point to the same object      3-->
        System.out.println(blueHouse.getColor());     // blue
        System.out.println(anotherHouse.getColor()); //blue

        anotherHouse.setColor("yellow");                4-->
        System.out.println(blueHouse.getColor())   //yellow
        System.out.println(anotherHouse.getColor());//yellow

      House greenHouse = new House("green");//dereference 5

     anotherHouse = greenHouse;                         6-->

     System.out.println(blueHouse.getColor());  //yellow
     System.out.println(greenHouse.getColor()); //green
     System.out.println(anotherHouse.getColor()); //green
  }
}
```

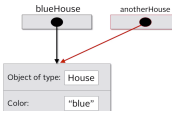
```
// House is blueprint
// instance variable(field) private String color;

class House {
   public House (String color) {
        this .color = color;
    }
    public String getColor() {
        return color;
    }
   public void setColor(String color) {
        this .color = color;
    }
}
```

1



1
This line creates a new instance of the House class. So remember house is a blueprint and more assigning it to the bluehouse variable, so in other words it's a reference to the object in memory.

2

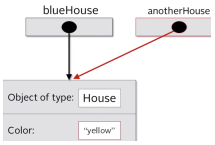


2
This line creates another reference to the same object in memory. There is still one house but two references to that one object.

3
Both will print blue

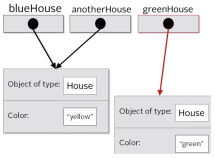
3
In this scenario, both will actually print blue and that's because again, we've got two references to the same object.

4



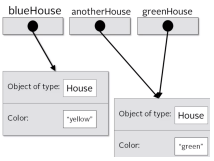
4
This line calling the method setColor and setting the color to yellow. Now to the left you can see that both blueHouse and anotherHouse now have the same color, Why?
Well, remember that we've got two references that point to the same object in memory. Now once we change the color of one, both references still point to the same object, so consequently they've both got the same value of yellow.

5



5
This line is creating another new instance of the house class, this time with the color set to green. So now we've got two objects in memory, but we've got three references (blueHouse, anotherHouse and greenHouse).
Now the variable or reference greenHouse points now to a different object in memory but blueHouse and anotherHouse point to the same object in memory at this point. So, this statement anotherHouse equals greenHouse, so we're assigning greenHouse to anotherHouse, so in other words, we're dereferencing another house and it will now point to a different object in memory.

6



6
finally we've got those last three print line statements, the first will print yellow since the blue house variable or reference points to the object in memory that has the yellow color, while the next two lines will print green and that's because anotherHouse in greenHouse point to the same object in memory.

THIS vs SUPER

- The keyword **super** is commonly used with method overriding, when we call/access a method with the same name from parent class.(variables and methods).
  - The only way to call a parent constructor is by calling **super()**.
  - A constructor can have a call to super() or this() but never both.
  - **super()** example;

```
class Shape {
   private int x;
   private int y;

   public Shape(int x, int y) {
       this.x = x;
       this.y = y;
   }
}

class Rectangle extends Shape {
   private int width;
   private int height;

   // 1st constructor
   public Rectangle(int x, int y) {
       this(x, y, 0, 0); // calls 2nd constructor
   }

   // 2nd constructor
   public Rectangle(int x, int y, int width, int height) {
       super(x, y); // calls constructor from parent (Shape)
       this.width = width;
       this.height = height;
   }
}
```

▪We have a class Shape width x,y variables and class Rectangle that extends Shape with variables width and height.

▪In Rectangle, the 1st constructor we are calling the 2nd constructor.

▪The 2nd constructor calls the parent constructor with parameters x and y

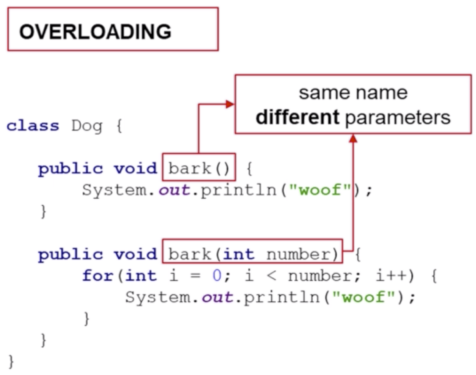
▪The parent constructor will initialize x, y variables while the 2nd Rectangle constructor will initialize width and height variables.

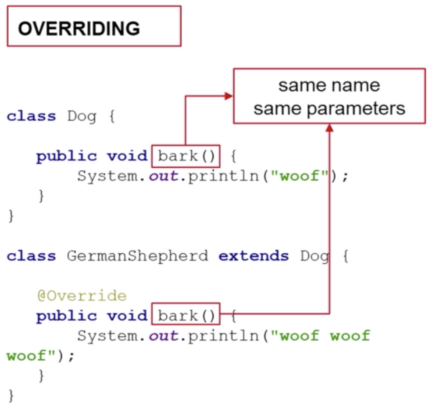
▪Here we have both super() and this() method calls.

- The keyword **this** is used to call the **current class** members(variables and methods). This is required when we have a parameter with the same name as instance variable(field). The keyword **this** is commonly used with constructors and setters. 2 types of this;
  - **this.**name = name;
  - **this();**
    → to call a constructor from another overloaded constructor in the same class.
    → It can be used only in a constructor, and it must be the first statement in a constructor.
    → It is used with constructor chaining, in other words when one constructor calls another constructor, and helps to reduce duplicated code.

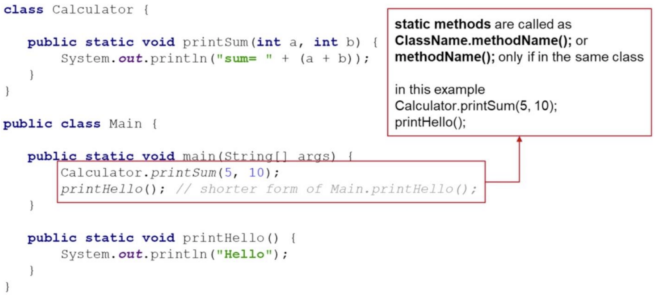
! ! ! We can use both of them anywhere in a class **except in static areas** which is the static block or a static method, any attempt to do so there will lead to compile time errors.

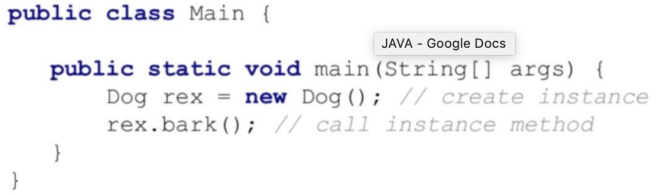
METHOD OVERLOADING	vs	METHOD OVERRIDING
• **Method overloading** means providing two or more separate methods in a class with the **same name** but **different parameters.** • Method return type may or may not be different and that allows us to **reuse** the same method name. • Java Developers often refer to overloading as *Compile Time Polymorphism*. • The compiler decides which method is going to be called based on the method name, return type, and argument list.		• **Method overriding** means defining a method in a child class that already exists in the parent class with same signature (same name, same arguments). • By extending the parent class the child class gets all the methods defined in the parent class • It is also known as *Runtime Polymorphism*, because the method that is going to be called is decided at runtime by JVM. • When we override a method it is recommended to put **@Override** immediately above the method definition. • Only inherited methods can be overridden, in other words methods can be overridden only in child class. • Constructors and private methods **CANNOT BE** overridden. • Methods that are final cannot be overridden. • A subclass can use **super.methodName()** to call the superclass version of an overridden method.





STATIC METHODS	vs	INSTANCE METHODS
• **Static methods** are declared using **static** modifier. • **Static methods CANNOT** access instance methods and instance variables directly. • They are usually used for operations that do not require any data from an instance of the class. • In **static methods** we can not use the **this** keyword. • Whenever you see a method that **doesn't use instance variables** that method should be declared as a **static method**. For example main is a static method and it is called by the JVM when it starts an application.		• Instance methods can access instance methods and instance variables directly. • Instance methods can also access static methods and static variables directly.





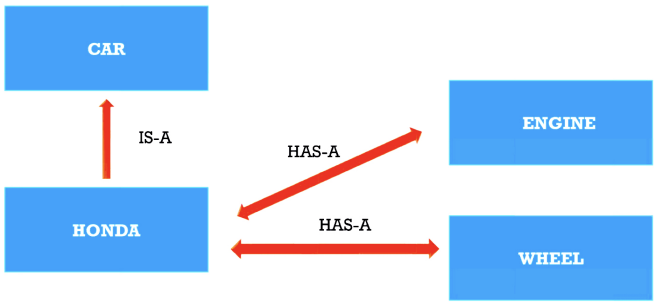
STATIC AND INSTANCE METHOD RECAP

```java
public class StaticMethods {

    public static void main (String[] args) {

        //static methods belongs to class
        //to reach static method in the Abc class
        Abc.show1();
        Abc.show4();

        //instance methods belongs to object
        //to reach instance method in the Abc class
        Abc x = new Abc();
        x.show2();
    }
}
```

```java
class Abc{
    public static void show1() {
        System.out.println("hi");
        show2();//error-> cannot reach from static to inst
        show4();  // we can reach from static to static
    }
    public void show2() { //instance method-> non-static
        System.out.println("hi");
        show1(); //we can call from instance to static
    }
    public void show3() {
        System.out.println("hi");
        show2(); //we can call from instance to instance
    }
    public static void show4() {
        System.out.println("hi");
    }
}
```

COMPOSITION

- Inheritance is a relationship.
- Composition has a relationship between objects.
- Object of one class is created as **data member** in another class. There is no specific implementation of “has-a” relationship but mostly we are depended upon “**new**” keyword.




```java
public class Car {
    private String color;
    private int maxSpeed;

    public void carInfo() {
        System.out.println("Car color: "+color+" ,Car maxspeed:
"+maxSpeed);
    }

    public Car(String color, int maxSpeed) {
        this.color = color;
        this.maxSpeed = maxSpeed;
    }

    public String getColor() {
        return color;
    }
    public int getMaxSpeed() {
        return maxSpeed;
    }
}
```

```java
//inheritance --> Honda extends Car --> Honda is a Car
public class Honda extends Car{

    //Composition --> Engine is a class and Honda has Engine
    private Engine engine;  //reference


    public Honda(String color, int maxSpeed, Engine engine) {
        super(color, maxSpeed);
        this.engine = engine;
    }

    public void startHonda() {
        engine.start();
        System.out.println("Honda started");
    }
}
```

```java
public class Engine {

    public void start() {
        System.out.println("Engine started");
    }

    public void stop() {
        System.out.println("Engine stopped");
    }
}
```

```java
public class Main {

  public static void main(String[] args) {

    Engine e1 = new Engine();

    //Honda h1 = new Honda("Red", 200, e1);
    Honda h1 = new Honda("Red", 200, new Engine());
    h1.carInfo();
    h1.startHonda();
  }
}
```

TERMINAL

Car color: Red ,Car maxspeed: 200
Engine started
Honda started

POLYMORPHISM

- Polymorphism means taking many forms. It is the ability of a variable, function, or object to take on multiple forms.

```java
public class Animal {
    public void speak() {
        System.out.println("Animals are speaking");
    }
}
```

```java
public class Dog extends Animal{
    @Override
    public void speak() {
        System.out.println("Dog is speaking");
    }
}
```

```java
public class Cat extends Animal{
    @Override
    public void speak() {
        System.out.println("Cat is speaking");
    }
}
```

```java
public class Bird extends Animal{
    @Override
    public void speak() {
        System.out.println("Bird is speaking");
    }
}
```

```java
public class Main {

    public static void main(String[] args) {

//      Cat c1 = new Cat();
//      c1.speak();          //Cat is speaking
//      Dog d1 = new Dog();
//      d1.speak();          //Dog is speaking
//      Bird b1 = new Bird();
//      b1.speak();          //Bird is speaking

        //polymorphism --> one object (a) can have
different forms

        Animal a = new Dog();  //object a--> reference to
Animal and goes to Dog class object
        a.speak();           //Dog is speaking

        a = new Cat();       //!!!garbage collection
-->After cat create, Dog is going to the garbage
        a.speak();           //Cat is speaking

        a = new Bird();
        a.speak();           //Bird is speaking
    }
}
```

	Overloading	**Overriding**
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method
Meaning	More than one method shares the same name in the class but having different signature	Method of base class is re-defined in the derived class having same signature
Behavior	To add/extend more to method's behavior	To change existing behavior of method
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature

ABSTRACTION

- Abstraction means we **focus on the essential** qualities of something rather than one specific example.
  - Focus on the essential (we know phone need to call(), but we don't know how do iphone, samsung call())
  - Ignore the irrelevant
  - Ignore the unimportant
- In Java, abstraction is achieved **by interfaces and abstract classes.** We can achieve 100% abstraction using interfaces.

1. ABSTRACT CLASSES

CREATING ABSTRACT CLASSES

- **abstract** keyword is used to create abstract class.
- An abstract class **can not** be instantiated.
- Goal is to provide reusable variables and methods to sub classes.

ABSTRACT CLASS RULES REVIEW

- It cannot be instantiated directly.
- It may be defined with any number, including zero, of abstract and non-abstract methods.
- It may not be marked as private or final.
- An abstract class that extends another abstract class inherits all of its abstract methods as its own abstract methods.
- The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods

```
public abstract class Student{

}
```

CREATING ABSTRACT METHODS

- **abstract** keyword is used to create abstract method.
- Abstract method **does not have body**, only have signature.

ABSTRACT METHOD RULES REVIEW

- It may only be defined in abstract classes.
- It may not be declared private or final.
- It must not provide a method body / implementation in the abstract class for which it is declared.
- Implementing an abstract method in a subclass follows the same rules for overriding a method.

```
public abstract class Student{
   public abstract void attendClass();

}
```

CREATING CONCRETE CLASS

- A subclass of abstract class is called concrete class
- A first concrete class must implement all inherited abstract methods

```
public abstract class Student{
   public abstract void attendClass();
}

public class LocalStudent extends Student{
   @Override
   public void attendClass(){
       System.out.println("attending in person");
   }
}
```

EXTENDING ANOTHER ABSTRACT CLASS

- An abstract class can extend another abstract class. If so it is optional to implement abstract methods from abstract super class.
- A first concrete class must implement all inherited abstract methods.

```
public abstract class LocalStudent extends Student{
   public void attendClass();
}
```

2. INTERFACE

- Contract between a class and outside world
- Provide set of abstract methods
- A class implements an interface
- The class provides the behaviors included in the interface
- Interface can also store constants

CREATING AN INTERFACE

```
public interface Teachable{
   public static final boolean STUDY_HARD = true;
   public abstract void canLean();
   public abstract void doHomework();
}
```

IMPLEMENTING AN INTERFACE

- A class can implements more than one interface.

```
public class Student implements Teachable, Dreamer{
        //implementation code
}
```

- If a class both extend a class and implement an interface, extends should come first then implements keyword

```
public class Student extends Person implements Teachable,
Dreamer{
    //implementation code
}
```

```
public interface Teachable{
   public static final boolean STUDY_HARD = true;

   public abstract void canLean();
   public abstract void doHomework();
}
   public class Student implements Teachable{

   @Override
   public void canLean(){
       //code
   }

   @Override
   public void doHomework(){
       //code
   }
}
```

INTERFACE RULES

- It is a abstract type and can not be instantiated. →

```
public interface Teachable{}
Teachable t = new Teachable(); //DOES NOT COMPILE
```

- An interface is abstract by default and may not be marked as final →

```
public final interface Teachable{} //DOES NOT COMPILE
```

- All fields in interface is automatically public static final even no declared such →

```
public final interface Teachable{
   boolean STUDY_HARD = true;
   void canLean();
   public abstract void doHomework();
}
```

EXTENDING ANOTHER INTERFACE

- An interface can extends another interface using **extends** keyword to share functionality.
- Unlike class, an interface can **extends multiple interfaces.**
- First concrete class has to implement all the abstract methods from both interface.

```
public interface Teachable{
   void canLearn();
}
public interface Mentorship extends Teachable,Bright{
   void mentor();
}
```

COMPOSITION

- Object of one class is created as **data member** in another class. There is no specific implementation of “has-a” relationship but mostly we are depended upon “**new**” keyword.

COLLECTION FRAMEWORK

- **What is the need for collections?**
  - I have 1000 variable to declare. How should I do that?

```
int var1 =10;
int var2 =20;
int var3 =30;
...
int var1000 =40;
```

- **Using Arrays**

```
int[] myArray = new int[1000];
```

- **Limitations with Array**
  - Fixed in Size
  - Homogenous data type

```
Object[] myArray = new Object[1000];
myArray[0]=1;
myArray[1]="Apple";
```

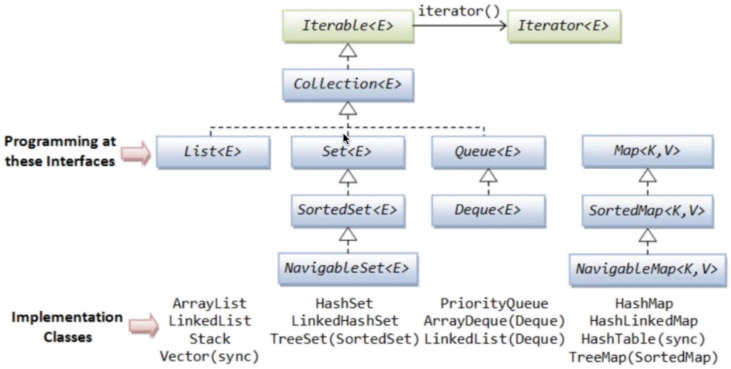
  - Arrays not implemented based on some standard data structure. There is no ready made methods. Programmer needs to write the methods like sort, isElementPresent, etc

- **COLLECTION**
  - Growable in nature. Can increase or decrease the size.
  - Can hold different data types.
  - Standard data structure. There are ready methods to use.
  - **Which concept is recommended to use?**

■ If you know the size in advanced, better to use arrays. ■ Consider performance issue while using the collections.	

	ARRAYS	COLLECTIONS
Size	Fixed in Size	Growable in Size
Point of Memory	−	+
Point of Performance	+	−
Data Type	Homogenous	Homogenous & Heterogenous
Data Structure	−	+
Can hold:	Primitive & Object Types	Only Object Types

  - **What is Collection?**
    - Collection is a group of individual objects as a single entity.
  - **What is Collection Framework?**
    - It defines several classes and interfaces which can be used to represent a group of objects as single entity.

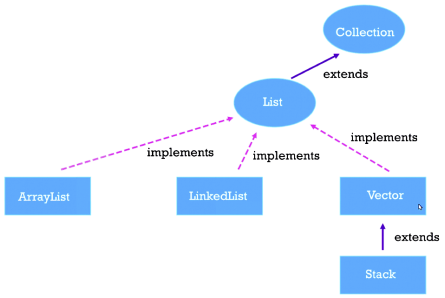


INTERFACES;

1. **COLLECTION**
   a. Collection interface defines the most common methods which are applicable for any collection object
   b. In general collection interface is considered as root interface of collection framework.
   c. There is no concrete class which implements collection interface directly.
   d. **Root interface** with basic methods like **add(), remove(), contains(), isEmpty(), addAll()**, ...etc

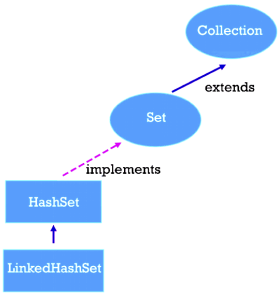
2. **LIST**

a. List is child interface of Collection.
b. If we want to represent a group of individual objects as a single entity where **duplicates are allowed** and **insertion order preserved** then we should go for List.



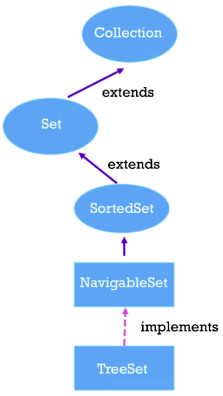
3. **SET**

a. Set is child interface of Collection.
b. If we want to represent a group of individual objects as a single entity where **duplicates are NOT allowed** and **insertion order NOT preserved** then we should go for List.



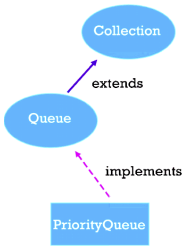
4. **SORTEDSET & 5. NAVIGABLESET**

a. SortedSet is child interface of Collection.
b. If we want to represent a group of individual objects as a single entity where **duplicates are NOT allowed** and **insertion order preserved** then we should go for SortedSet.
c. NavigableSet is child interface of SortedSet and it defines several methods for navigation purposes.



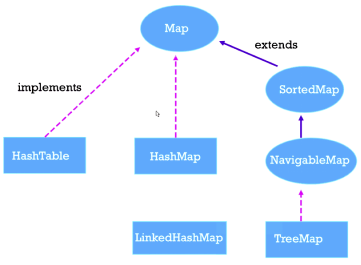
6. **QUEUE**

a. It is child interface of Collection.
b. If we want to represent a group of individual objects prior to processing then we should go for Queue.



7. **MAP & 8. SORTEDMAP & 9. NAVIGABLEMAP**

a. It is child interface of Collection.
b. Key and Value, we will use mostly.



ArrayList Review

```java
import java.util.*;

public class GroceryList {

    //we are using ArrayList because we don't know the size of our
groceryList
    //if I have interface and concrete class
    //List groceryList = new ArrayList();     OR
    List <String> groceryList = new ArrayList<String>();

    //to add item to my ArrayList
    public void addGroceryItem(String item) {
        groceryList.add(item);
    }

    //to print items in my List
    public void printGroceryList() {
        System.out.println("You have " + groceryList.size() + "
items in your grocery list");
        for (int i = 0; i < groceryList.size(); i++) {
            System.out.println((i+1)+"."+ groceryList.get(i));
        }
    }

    //to modify my list, change one item with another item
    public void modifyGroceryItem(int position, String newItem) {
        groceryList.set(position, newItem);
    }

    //to remove something
    public void removeGroceryItem(int position) {
        //String theItem = groceryList.get(position);
        groceryList.remove(position);
    }

    //to find item
    public String findItem(String searchItem) {
     //boolean exists=groceryList.contains(searchItem);->one method
        int position = groceryList.indexOf(searchItem);
        if (position>=0) {
            return groceryList.get(position);
        }
        return null;
    }
}
```

```java
import java.util.Scanner;
public class Main {
    private static Scanner scanner = new Scanner(System.in);
    private static GroceryList groceryList = new GroceryList();
    public static void main(String[] args) {
        boolean quit = false;
        int choice = 0;
        printInstructions();

        while(!quit) {
            System.out.println("Enter your choice: ");
            choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
            case 0:
                printInstructions();
                break;
            case 1:
                groceryList.printGroceryList();
                break;
            case 2:
                addItem();
                break;
            case 3:
                modifyItem();
                break;
            case 4:
                removeItem();
                break;
            case 5:
                searchItem();
                break;
            case 6:
                quit = true;
                break;
            }
        }
    }
    public static void printInstructions(){
        System.out.println("0:Print Instructions\n1:Print Grocery
list\n2:Add\n3:Modify \n4:Remove\n5:Search \n6:Quit");
    }
    public static void addItem(){
        System.out.println("Add ITEM ");
        String addItem=scanner.nextLine();
        groceryList.addGroceryItem(addItem);
    }
    public static void modifyItem(){
        System.out.println("Modify please enter number of grocery ");
        int position=scanner.nextInt();
        scanner.nextLine();
        System.out.println("Modify please enter new item ");
        String newItem=scanner.nextLine();
        groceryList.modifyGroceryItem(position-1,newItem);
    }
    public static void removeItem(){
        System.out.println("Please enter grocery to remove");
        int position=scanner.nextInt();
        groceryList.removeGroceryItem(position-1);
    }
    public static void searchItem(){
        System.out.println("Please enter item to search");
        String item=scanner.nextLine();
        String position=groceryList.findItem(item);
        System.out.println("Searched item returned value "+position);
    }
}
```

Collection Framework Review

1- LIST

ArrayListClass.java

```
package List;
import java.util.*;

public class ArrayListClass {

    public static void main(String[] args) {

        List<Integer> values = new
ArrayList<Integer>();
        values.add(5);
        values.add(10);
        values.add(1,9);   // -> it replace
the 1th item with 9
        values.add(2);

        Collections.sort(values);

        System.out.println(values);    //
[5,9,10]

        for(int i : values) {
            System.out.println(i);
        }
    }
}
```

```
//
[2, 5, 9, 10]
2
5
9
10
```

LinkedListClass.java
--> **work with NOTES (previous and next)**

```
package List;
import java.util.LinkedList;

public class LinkedListClass {

    public static void main(String[] args) {

        LinkedList<String> object = new
LinkedList<String>();

        object.add("A");
        object.add("B");
        object.addLast("C");
        object.addFirst("D");
        object.add(2, "E");
        object.add("F");
        object.add("G");
        System.out.println(object);

        object.remove("B");
        object.remove(3);
        object.removeFirst();
        object.removeLast();
        System.out.println(object);

        int size = object.size();
        System.out.println(size);

        System.out.println(object.get(2));
    }
}
```

```
[D, A, E, B, C, F, G]
[A, E, F]
3
F
```

VectorListClass.java

```
package List;
import java.util.Vector;

public class VectorListClass {

    public static void main(String[] args) {

        //Vector has an initial size of 10;
and the increase capacity is 100%
        //Vector is slow -  not good to use
        Vector<Integer> v= new
Vector<Integer>();

        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);
        v.add(1);

        v.add(2);

        System.out.println(v.capacity());
    }
}
```

```
//
20
```

2- SET

HashSetClass.java

```
package Set;

import java.util.HashSet;
import java.util.Set;

public class HashSetClass {

    public static void main(String[] args) {

        Set<Integer> values = new HashSet<Integer>();
        values.add(15);
        values.add(19);
        values.add(6);

        values.add(6);
        values.add(8);
        values.add(88);

        for(int i: values) {
            System.out.println(i);  //19, 6 , 8 , 88, 15
            //insertion order NOT preserved
            //duplicates are NOT allowed
        }
    }
}
```

TreeSetClass.java

```
package Set;

import java.util.Set;
import java.util.TreeSet;

public class TreeSetClass {

    public static void main(String[] args) {

        Set<Integer> values = new TreeSet<Integer>();
        values.add(15);
        values.add(19);
        values.add(6);

        values.add(6);
        values.add(8);
        values.add(88);

        for(int i: values) {
            System.out.println(i);  //6, 8, 15, 19, 88
            //insertion order preserved
            //duplicates are NOT allowed
        }
    }
}
```

3-QUEUE

```
package Queue;
import java.util.*;

public class QueueClass {

    public static void main(String[] args) {

        Queue<Integer> q = new PriorityQueue<Integer>();

        // Add elements
        for(int i=0; i<5;i++) {
            q.add(i);
        }

        System.out.println(q); //[0, 1, 2, 3, 4]

        int removedel = q.remove();  //it is removing
first element
        System.out.println(removedel); // 0

        System.out.println(q);         // ->[1, 3, 2, 4]
        System.out.println(q.peek()); // ->1 -> picking
first one
        System.out.println(q.size()); // ->4
    }
}
```

4- MAP → (We will use it in Selenium)

```
//HashMap-fast, unsynchronized, works with single thread, allows one null key
//HashTable-slow, synchronized, works with multiple thread, NOT allows one
null key
//LinkedHashMap- preserves the insertion order

package Map;
import java.util.*;

public class HashMapClass {
    public static void main(String[] args) {
        //Map map = new HashMap();
        //If I want to specify;
        Map<String,String> map = new HashMap<>();

        map.put("myName", "Mike");
        map.put("myJob", "Developer");
        map.put("myAge", "25");

        System.out.println(map);//{myName=Mike, myJob=Developer, myAge=25}
        System.out.println(map.get("myName")); //Mike

        Set<String> keys = map.keySet();
        for (String key: keys) {
            System.out.println(map.get(key)); //Mike , Developer, 25
        }
    }
}
```

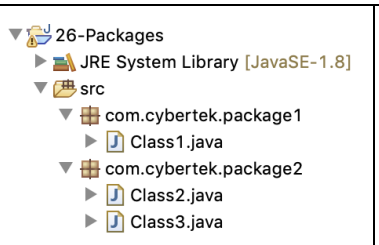
STRING MANIPULATION

```
public class Main {

    public static void main(String[] args) {

        //charAt(index) -> returns char value for the particular index
        String str = "JavaScript";
        System.out.println(str.charAt(3));  //a

        //length -> returns string length
        System.out.println(str.length());   //10

        //substring(int beginIndex) -> returns substring for given begin index
        //substring(int beginIndex, int endIndex) -> returns substring for given begin and end index
        System.out.println(str.substring(5));   //cript
        System.out.println(str.substring(5, 7));//cr

        //contains() -> returns true / false after matching the sequence of char value
        System.out.println(str.contains("va2")); //false

        //equals(Object another) -> checks the equality of string with the given object
        System.out.println(str.equals("Java")); //false

        //isEmpty() -> check if tring is empty
        System.out.println(str.isEmpty());  //false

        //concat(string str) -> concatenates the specified string
        System.out.println(str.concat("batch9")); //JavaScriptbatch9

        //replace(char old, char new) -> replace all occurrences of the specified char value
        System.out.println(str.replace("Script", "")); //Java

        //equalsIgnoreCase(string another) -> compares another string. It does not check case
        System.out.println(str.equalsIgnoreCase("javascript"));  //true

        //split(String delimeter) -> returns a split matching delimiter
        String str2 ="JavaScript is the best batch ever";
        String[] myArray = str2.split(" ");
        for(String word:myArray) {
            System.out.println(word);
        }

        //indexOf(int ch) -> returns the specified char value index
        System.out.println(str.indexOf("v")); //2

        //indexOf(String substring, int fromIndex) -> returns specified substring index starting with given index
        System.out.println(str.indexOf("a", 4)); //-1

        //toLowerCase() -> returns a string in lowercase
        //toUpperCase() -> returns a string in uppercase
        System.out.println(str.toLowerCase());  //javascript
        System.out.println(str.toUpperCase());  //JAVASCRIPT

        //trim() -> removes beginning and ending spaces of this string
        String str3 = "JavaScript     ";
        System.out.println(str3.trim());  //JavaScript

    }
}
```

DATE

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        //SimpleDate format is a concrete class for formatting and parsing date which inherits
        // java.text.Dateformat class

        //FORMATTING -> converting date to string
        Date date = new Date();
        System.out.println(date);  //today's date -> Wed Nov 21 11:18:33 EST 2018

        SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy"); //M->month, m->minute
        String strDate = formatter.format(date);
        System.out.println(strDate);  // 21/11/2018

        formatter = new SimpleDateFormat("MM/dd/yyyy");
        strDate = formatter.format(date);
        System.out.println(strDate);  //11/21/2018

        formatter = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
        strDate = formatter.format(date);
        System.out.println(strDate);  //21-11-2018 11:25:46

        formatter = new SimpleDateFormat("dd MMMM yyyy zzzz");
        strDate = formatter.format(date);
        System.out.println(strDate);  //21 November 2018 Eastern Standard Time

        formatter = new SimpleDateFormat("E, d MMM y H:m:s z");
        strDate = formatter.format(date);
        System.out.println(strDate);  //Wed, 21 Nov 2018 11:31:12 EST

        //PARSING -> converting string to date
        formatter = new SimpleDateFormat("dd/MM/yyyy");
        try {
            date = formatter.parse("31/03/2015");
        } catch (ParseException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println(date);  //Tue Mar 31 00:00:00 EDT 2015

    }
}
```

PACKAGES




```
package com.cybertek.package1;

//import com.cybertek.package2.*;       // 1st way
//import com.cybertek.package2.Class2;  // 2nd way

public class Class1 {

    public static void main(String[] args) {
        /*   3 ways to access package from outside the package
            1- import package.*;
            2- import package.className;
            3- fully qualified name
        */
        //Class2 cl = new Class2();
        //Class3 cl1 = new Class3();

        com.cybertek.package2.Class2 cl = new com.cybertek.package2.Class2(); //3rd
way
    }
}
```

ACCESS MODIFIERS

- There are 4 access modifiers available in Java

public	● Private members are accessible: ○ Within the defining class ○ In another class in the same package as the defining class ○ In a class defined in another package
private	● Private members are accessible to ○ **only within the defining class**
default(do not specify anything)	● Default members are accessible to ○ Within the defining class ○ In another class in the same package as the defining class
Protected	● The protected access modifier is accessible within package and outside the package but through **inheritance only.** ● The protected access modifier can be applied on the data member, method and constructor. **It can't be applied on the class.**

- Top level Java class can have two access modifiers: public and default
- Variables, Constructors and methods can have all four access modifiers.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

EXAMPLE

A.java

```
package com.cybertek.p1;

public class A {
    public static void display() {
        System.out.println("Hello");
    }
}
```

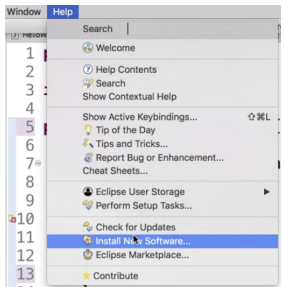
B.java

```
package com.cybertek.p2;
import com.cybertek.p1.A;

public class B{

    public static void main(String[] args) {

        A object = new A();
        object.display();
    }
}
```

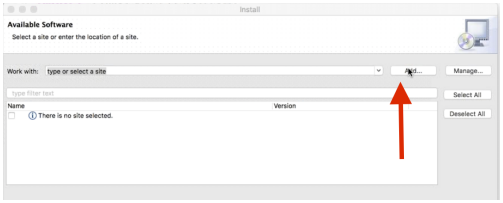
```
//PROTECTED EXAMPLE

package com.cybertek.p1;

public class A {
    protected static void display() {
        System.out.println("Hello");
    }
}
```

```
//PROTECTED EXAMPLE

package com.cybertek.p2;
import com.cybertek.p1.A;

public class B extends A{
    public static void main(String[] args) {
        A object = new A();
        object.display();
    }
}
```

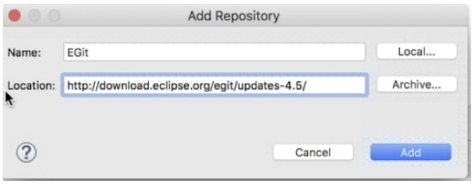
GITHUB CONNECTION



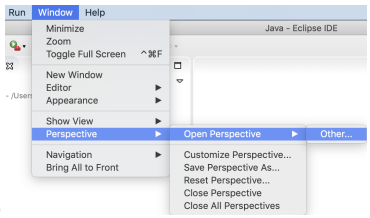
1



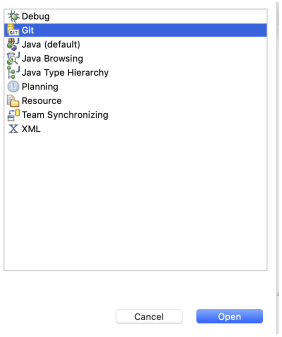
2



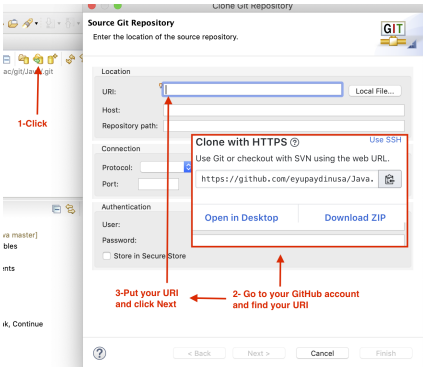
3



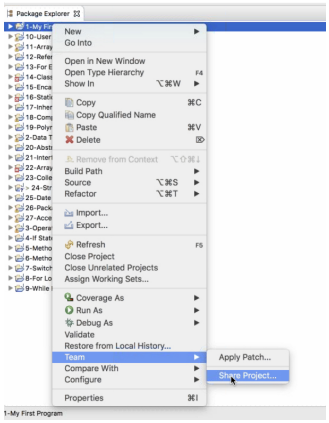
4



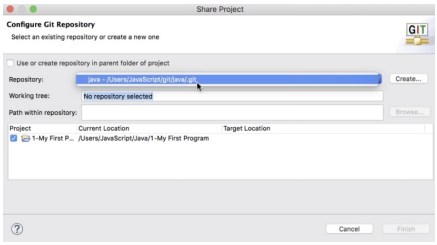
5



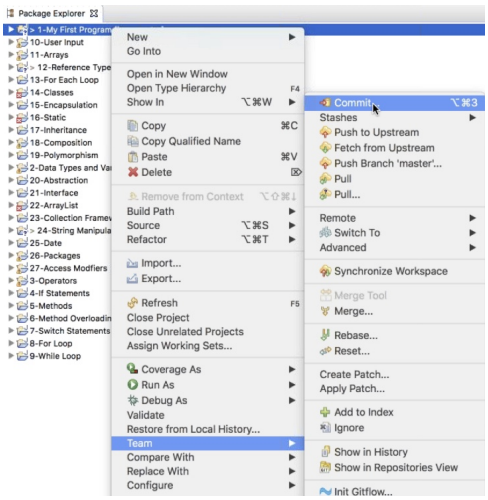
6



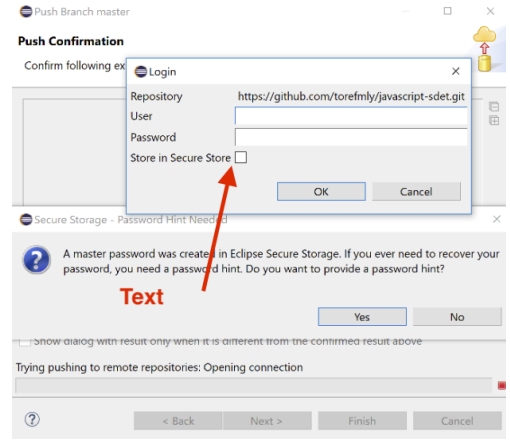
7



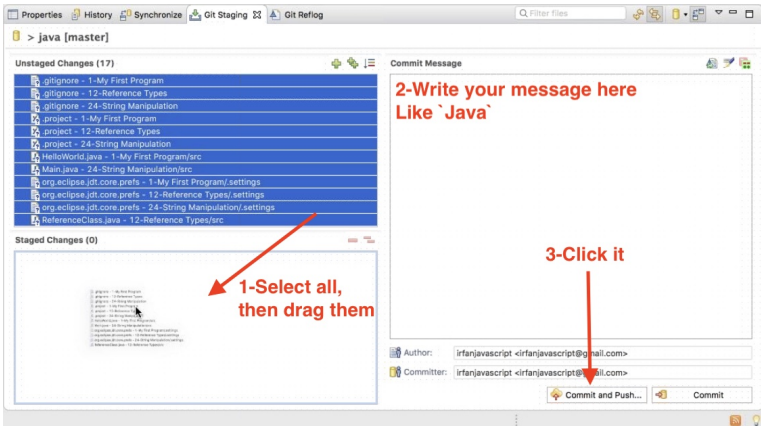
8



9



10



11

12- You can check your github to make sure that your file uploaded.