Campus Corners

Scratch2Corporate...

Java

NIKHIL B. DETHE

DATA SCIENTIST

Programming

- Program is a set of instructions that you give to a computer so that it will do a particular task.
- A **program** (noun) is an executable **software** that runs on a computer.
- **Programming : “Instruct the computer”**, this basically means that you provide the computer a set of instructions that are written in a language that the computer can understand.
- **Programming** is the process of creating a set of instructions that tell a computer how to perform a task.
- **Programming** can be done using a variety of computer **programming** languages, such as C, C++, Java, Python, C#, VB and JavaScript.

C, C++ & Java

C	C++	JAVA
Procedural language	Object-Oriented Programming (OOP)	Pure Object-Oriented Programming (OOP)
Developed by Dennis Ritchie in 1972	Developed by Bjarne Stroustrup in 1979	Developed by James Gosling in 1991
Platform Dependent	Platform Dependent	Platform Independent
Pointers are supported	Pointers are supported	Pointers are not supported
Compiler generates .exe file	Compiler generates .exe file	Compiler generates .class file
Uses malloc and calloc for memory	Uses new and delete for memory	Uses new and garbage collector for memory
Inheritance is not supported	Inheritance is supported	Inheritance is supported except Multiple inheritance
Multi-threading is not supported	Multi-threading is not supported	Multi-threading is supported
Database connectivity is not supported	Database connectivity is not supported	Database connectivity is supported
Exception handling is not supported	Exception handling is supported	Exception handling is supported

History of Java

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
- Initially designed for small, embedded systems in electronic appliances like set-top boxes.
- Firstly, it was called "Greentalk" by **James Gosling**, and the file extension was .gt.
- After that, it was called Oak and was developed as a part of the Green project.
- In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.
- Java was owned by Sun Microsystems.
- In 2010 Oracle Corporation took over Sun Microsystems.

Features of Java

- Simple
- Object Oriented
- Portable
- Platform-independent
- Robust
- Secured
- Multi threaded
- Architecture neutral
- High Performance
- Interpreted
- Dynamic
- Distributed
- Open Source
- Scalable

Lets start…

Q. Write a program to print ‘Hello World!’ in C, C++ and Java.

Hello World in C & C++

C

```
#include <stdio.h>

main() {

     printf(“Hello World!”);

}
```

C++

```
#include <iostream.h>

void main() {

     cout << “Hello World!”;

}
```

Hello World in Java

```
class HelloWorld {
   public static void main(String args[]){
         System.out.println(“Hello World”);
    }
}
```

Keyword

- Keywords are special types of words which have some **predefined meaning** in programming.
- Java language have **52 keywords**.
- We cannot use keywords as an identifiers.
- Conventions : All keywords are in **lowercase**.
- In Java **true, false** & **null** are reserved words, but they are not keywords, instead they are **literals**.

Hello World

keywords

```
class HelloWorld {
   public static void main(String args[]){
      System.out.println(“Hello World”);
   }
}
```

Keyword class

- class keyword is used to create a named class.
- class keyword is followed by the name of the class.
- In this example we are creating a class with name HelloWorld.

Identifier

- Identifier is the name assign to either class, variable, object, interface, etc.
- Identifiers are used to identify a particular class, variable, object, etc.
- e.g.

int len;   Box obj;   class Demo { }   int arr[];   interface intf { }

In the above examples **len** is an variable identifier, **obj** is an object identifier of type Box, **Demo** is a class identifier, **arr** is an array identifier of type Integer, and **intf** is an interface identifier.

Rules for Identifiers

- We can use alphanumeric characters i.e. [A-Z], [a-z] and [0-9] while naming identifiers.
- We cannot use special characters while naming identifiers except $(dollar) and _ (underscore) in identifiers.
- We cannot use keywords as an identifiers.
- Java identifiers are case-sensitive.
- Java identifier cannot start with a digit or an $.

Hello World

HelloWorld

```
class HelloWorld {
    public static void main(String args[]){
        System.out.println(“Hello World”);
    }
}
```

In the above example HelloWorld is a class Identifier.

Access Modifiers

- Access modifiers specifies the scope or visibility of the member. It is also called as access specifiers.
- There are four access modifiers or access specifiers in java i.e. public, private, protected and default.
- If we don’t provide any access modifier to a method, variable or an object explicitly then it posses default access modifier by default.

Access Modifiers

1) **private :** This is also called as local scope. Any member declared with this access specifier can be accessed only from the same class.

2) **public :** This is also called as global scope. Any member declared with this access specifier can be accessed from any class.

3) **protected :** This is also called as inherited scope. Any member declared with this access specifier can be accessed from any class in same package or any sub class from different package.

4) **default :** This is also called as packaged scope. Any member with this access specifier can be accessed from any class in same package.

Access Modifiers

	Same class	Non-sub class in same package	Sub class in same package	Non-sub class in different package	Sub class in different package
private	✓	✗	✗	✗	✗
public	✓	✓	✓	✓	✓
protected	✓	✓	✓	✗	✓
default	✓	✓	✓	✗	✗

Hello World

public

```
class HelloWorld {
     public static void main(String args[]){
          System.out.println(“Hello World”);
     }

}
```

In the above example we have declared main method as public so that it can be accessed from anywhere.

Non-Access Modifiers

- Non-Access modifiers do not control access level but provides other functionality.
- Following are the Non-Access modifiers in java.
    - static
    - final
    - abstract
    - transient
    - volatile
    - synchronized

Types of variables in Java

There are three types of variables in Java, based on where they are declared.

1) local variable
2) instance variable
3) class variable

```
class MyDemo {
       int a;
       static int b;
       public void show() {
              int c;
       }
}
```

In this example **a** is an instance variable,

b is a class variable and **c** is a local variable.

local variables

- All the members declared inside a scope other than class scope are called as local variables.
- These variables are accessible only inside the scope where they are declared.
- Life cycle of these variable starts from the point where they are declared till the end of the scope where they are declared.
- In the previous example variable **c** is the local variable, because it is declared inside the method show().
- This variable is accessible only inside show() method.
- Each time when we call show() method, a new copy of **c** variable is created, and it goes out of scope when the closing curly bracket of method show() is encountered.

instance variables

- All the members declared inside a class scope and which are non-static in nature are called as instance variables.
- To access these variables we required object(instance) of that class.
- All the instance variables posses a separate copy for each object(instance), i.e. all the instance variables belongs to a particular object(instance).
- In the previous example variable **a** is an instance variable. To access this variable we require object of MyDemo.
- Each object of MyDemo class will posses a different copy for this variable.

```
e.g.    MyDemo obj1 = new MyDemo();
obj1.a = 111;
```

class variables

- All the members declared inside a class scope and which are static in nature are called as class variables.
- To access these variables we don’t required object(instance) of that class, we can access them with the help of class name.
- All the class variables posses a single shared copy for each object(instance), i.e. a single copy of class variables is shared among all the objects.
- In the previous example variable **b** is a class variable. To access this variable we don’t require object of MyDemo instead we can access it with class name MyDemo.
- Each object of MyDemo class will share a a different copy of this variable.

  e.g.   MyDemo.b = 999;

Hello World

static

```
class HelloWorld {
     public static void main(String args[]){
          System.out.println(“Hello World”);
     }
}
```

In the above example we have declared main method as static so that it can be accessed without any object(instance) or by using class name, i.e.

```
     HelloWorld.main(…)
```

Return Type

Whenever a method is returning a value its data type or type is explicitly specified in the method signature, it is called as return type.

In this example method add is returning ans so is data type i.e. int is specified in method signature.

Whenever a method is not returning anything

Then we use void keyword as a return type.

void return type means returning nothing.

```
public int add() {
        int ans;
        .........
        return ans;
}
```

Hello World

void

```
class HelloWorld {
    public static void main(String args[]){
        System.out.println(“Hello World”);
    }
}
```

In the above example we have used void as a return type, as main method is not returning anything.

Hello World

main

```
class HelloWorld {
     public static void main(String args[]){
          System.out.println(“Hello World”);
     }
}
```

In the above example main is the method identifier.

In java execution of a program starts from main method.

String args[] is a command line arguments.

Hello World

System.out.println(“Hello World!”);

```
System.out.println(“Hello World!”);
```

```
package java.lang;

public final class System {

    public static PrintStream out;

    .............

}
```

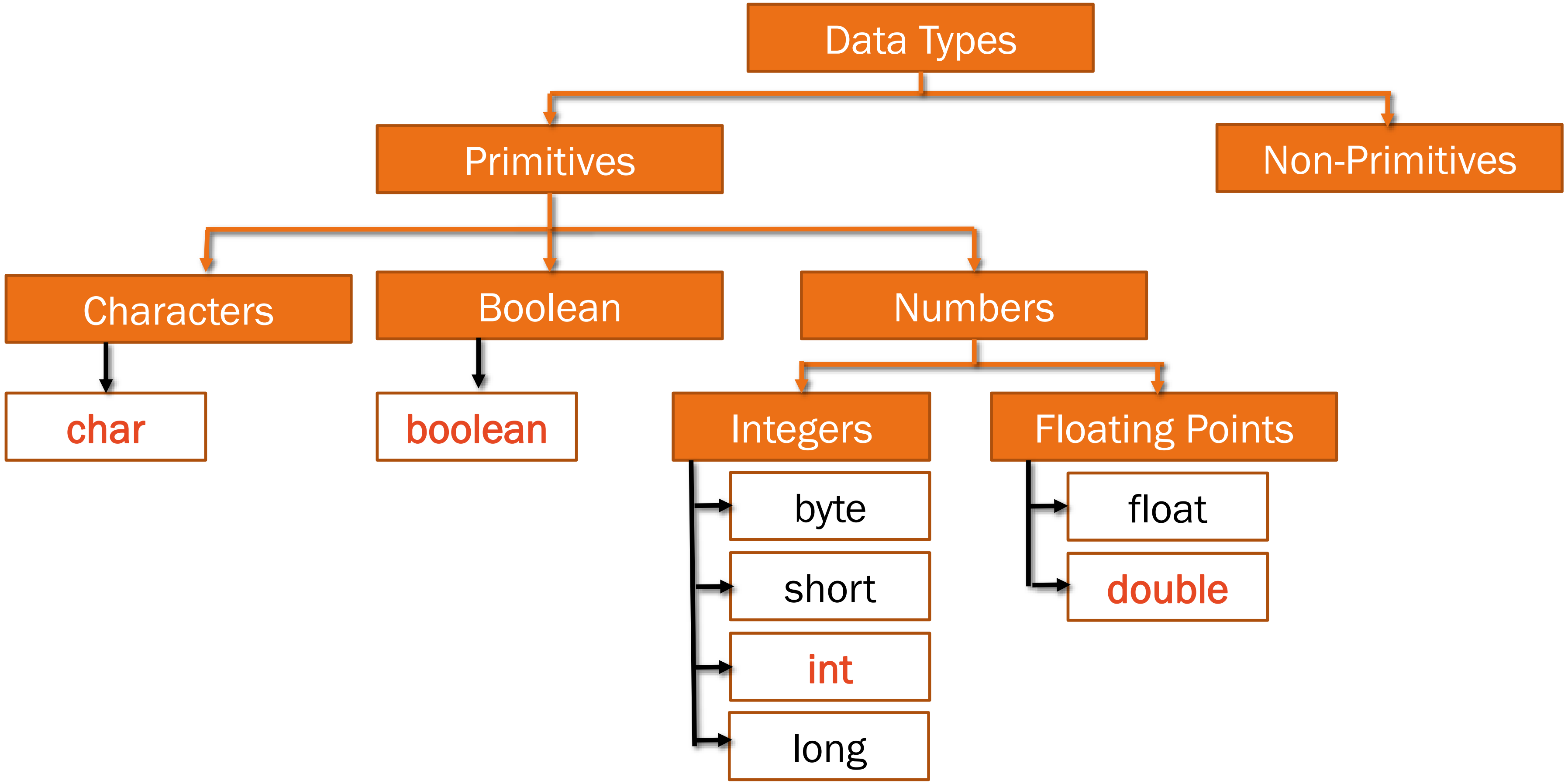
```
package java.io;

public class PrintStream {

public void println(String str)  {

            .............
           logic to print on command prompt
      }

}
```

To print ‘Hello World!’ we have to pass this string literal to the method println method of PrintStream class. This is a public non-static method so we can access it with the help of instance of PrintStream class, i.e. obj.println(“Hello World!”); where obj is object of PrintStream class. The instance of PrintStream class is already created in System class with the reference-variable out. This reference variable out is public and static so we can access it with the help of class name. As it is declared in class System, we can do System.out.

Data Types

- Every variable in Java should be of some specific type.
- This specific type is called as datatype.
- There are 2 types of datatypes in Java i.e. **value types** (primitives) & **reference types** (non-primitives).
- Value types are of **fixed size** so they are stored on a **stack memory**, and hence are **fast** to access.
- Reference types are of **variable size** so they are stored on a **heap memory**, and hence are **slow** to access.
- Whenever we assign a value type to another value type their **copy gets assigned**.
- Whenever we assign a reference type to another reference type then their **reference gets assigned**, it means they both point to a same value.

Data Types


Data Types
Primitives
Non-Primitives
Characters
Boolean
Numbers
char
boolean
Integers
Floating Points
byte
short
int
long
float
double

Data Types

Sr. No.	Data type	Disk space	Range	Default value
1.	byte	1 byte	Stores whole numbers. **-128** to **127**	0b
2.	short	2 bytes	Stores whole numbers. **-32,768** to **32,767**	0s
3.	int	4 bytes	Stores whole numbers. **-2,147,483,648** to **2,147,483,647**	0
4.	long	8 bytes	Stores whole numbers. **-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**	0*l*
5.	float	4 bytes	Stores fractional numbers. Can store up to **7 decimal digits**.	0.0f
6.	double	8 bytes	Stores fractional numbers. Can store up to **15 decimal digits.**	0.0
7.	boolean	1 bit	Can store only boolean values, i.e. **true** or **false.**	false
8.	char	2 bytes	Stores a single **character** or **ASCII values**.	'\u0000'

Literals & Variables

- Literals are constant values used in either expressions or assigned to variables.
- Variables are the named containers which holds the value.
- Each variable in java must have a specific type.
- This type decides the memory size and the kind of operations we can perform on them.
- There are 3 types of variables in Java.
    1) **Instance variable :** They are initialized to their default values.
    2) **Class variable (Static variable) :** They are initialized to their default values.
    3) **Local variable :** They don't get initialized to their default values, so we have to initialized them explicitly before using them.

Declaration of variables

```
int a;
```

Declared variable ‘a’ of type int. If it is a class variable or an instance variable then its default value is 0.

```
a = 45;
```

Variable ‘a’ is initialized with the value 45.

```
int a = 45;
```

Here variable ‘a’ is declared as integer and initialized with value 45.

```
int b = a;
```

Here variable ‘b’ is assigned a with copy of variable ‘a’.

Literals & Variables

examples

- byte
  - e.g. 12b -34b 0b -56b
  - byte temperature = 32b;
  - byte var1, var2 = -30b, var3;
- short
  - e.g. 12s -34s 0s -56s
  - short totalMarks = 32s;
  - short var1, var2 = -30s, var3;
- int
  - e.g. 12 -34 0 -56
  - int rollNo = 32;
  - int var1, var2 = -30, var3;
- long
  - e.g. 12*l* -34*l* 0*l* -56*l*
  - long distance = 32*l*;
  - long var1, var2 = -30*l*, var3;
- float
  - e.g. 12.56f -34.78f 0f 0.56f
  - float percentage = 69.53f;
  - float var1, var2 = -30.45f, var3;
- double
  - e.g. 12.56 -34.78 0 -0.56
  - double interestAmt = 5645.67;
  - double var1, var2 = -35.65, var3;
- boolean
  - e.g. true false
  - boolean flag = true;
  - boolean var1, var2 = true, var3;
- char
  - e.g. ‘a’ ‘n’
  - char gender =‘m’;
  - char var1, var2 =‘f’, var3;

Type Casting

- Converting one datatype into another is known as type casting or, type-conversion.
- If you want to store an ‘int' value into a simple ‘short’ then you can type cast ‘int' to ‘short’.
- When we want to convert a smaller **type** to the larger **type** then we use **implicit type casting**, it is also called as broadening.
- When we want to convert a larger **type** to the smaller **type** then we use **explicit type casting**, it is also called as narrowing.

Implicit Type Casting

```
short b = 45s;

int a = b;      // implicit type casting
```

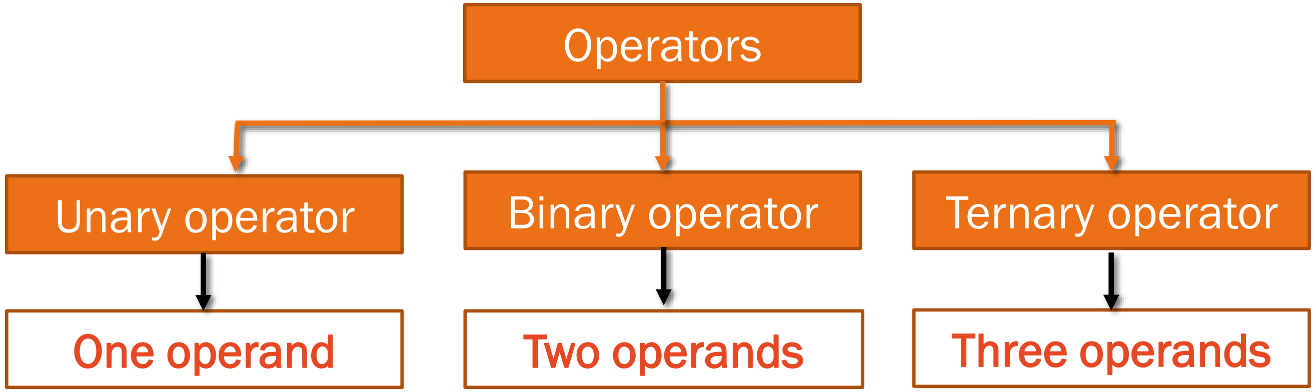
Explicit Type Casting

```
int a = 45;

short b = (short) a;  // explicit type casting
```

Operators

- Operators are used to perform operations on variables and values.
- Operators can be classified in two ways, i.e. based on operands used and based on functionality.
- Based on operands used, operators can be classified as
  1) unary operator
  2) binary operator
  3) ternary operator.
- Based on functionality, operators can be classified as
  1) Arithmetic operators
  2) Relational operators
  3) Increment / Decrement operators
  4) Logical operators
  5) Assignment operators
  6) Conditional operator
  7) Bitwise operators

Operators



Operators

Unary operator

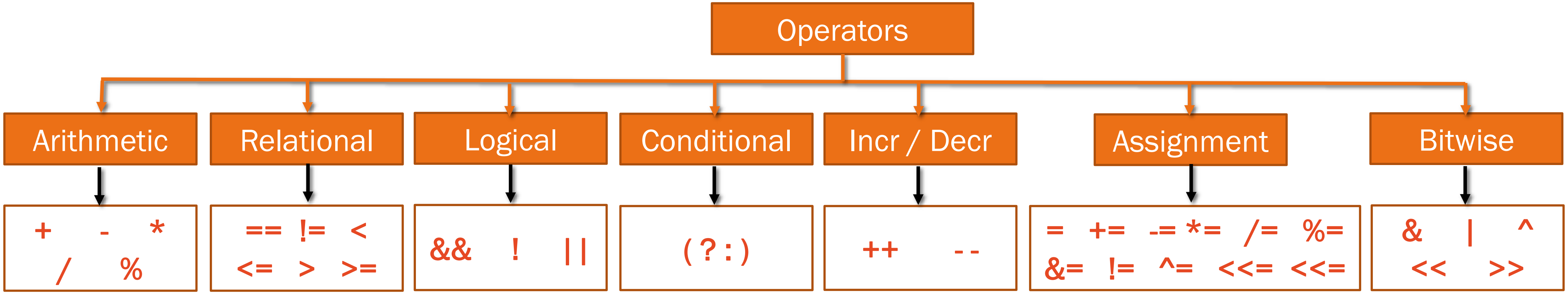
One operand

Binary operator

Two operands

Ternary operator

Three operands



Operators

Arithmetic

+ - * / %

Relational

== != < <= > >=

Logical

&& ! | |

Conditional

(? :)

Incr / Decr

++ - -

Assignment

= += -= *= /= %= &= != ^= <<= <<=

Bitwise

& | ^ << >>

Arithmetic operators

examples

+

```
int a = 20, b = 10;
int c = a + b;  // c = 30
int d = a + 80; // d = 100
```

-

```
int e = a - b;  // e = 10
int f = a - 5;  // f = 15
```

*

```
int g = a * b;  // g = 200
int h = a * 7;  // h = 140
```

/

```
int i = a / b;      // i = 2
int j = a / 7;      // j = 2
double k = a / 7;   // k = 2.857142
```

%

```
int m = a % b;      // m = 0
int n = a % 3;      // n = 2
int p = a % 7;      // p = 6
```

Relational operators

examples

==

```
int a = 20, b = 10, c = 20;
boolean x = (a == b); // x = false
boolean x = (a == c); // x = true
```

<

```
boolean x = (a < b); // x = false
boolean x = (b < a); // x = true
```

<=

```
boolean x = (a <= b); // x = false
boolean x = (b <= a); // x = true
boolean x = (a <= c); // x = true
```

!=

```
boolean x = (a != b); // x = true
boolean x = (a != c); // x = false
```

>

```
boolean x = (a > b); // x = true
boolean x = (b > a); // x = false
```

>=

```
boolean x = (a >= b); // x = true
boolean x = (b >= a); // x = false
boolean x = (a >= c); // x = true
```

Logical operators

! (Logical Not)

a	! a
true	false
false	true

|| (Logical Or)

a	b	OP
false	false	false
false	true	true
true	false	true
true	true	true

&& (Logical And)

a	b	OP
false	false	false
false	true	false
true	false	false
true	true	true

examples

```
boolean  a=true, b=false, c=true, d=false;

x = (!a);                // x = false

x = (!b);                // x = true

x = (a || b);            // x = true

x = (a || c);            // x = true

x = (a || d);            // x = true

x = (b || d);            // x = false

x = (a && b);            // x = false

x = (a && c);            // x = true

x = (a && d);            // x = false

x = (b && d);            // x = false
```

Conditional operator

examples

(condition) ? (expression 1) : (expression 2);

If the condition yields true then expression 1 is executed otherwise expression 2 is executed.

```
int a = 4, b = 5;
String str1 = (a%2 == 0)?(“Even”):(“Odd”);
String str2 = (b%2 == 0)?(“Even”):(“Odd”);
System.out.println(str1);         // Even
System.out.println(str2);         // Odd
```

Increment/Decrement operators

- examples

```
int a = 10;
```

Pre-Increment

```
System.out.println(a);    //     10
System.out.println(++a);  //     11
System.out.println(a);    //     11
```

Pre-Decrement

```
System.out.println(a);    //     10
System.out.println(--a);  //     9
System.out.println(a);    //     9
```

```
int a = 10;
```

Post-Increment

```
System.out.println(a);    //     10
System.out.println(a++);  //     10
System.out.println(a);    //     11
```

Post-Decrement

```
System.out.println(a);    //     10
System.out.println(a--);  //     10
System.out.println(a);    //     9
```

Bitwise operators

examples

```
int a = 240;       // a = 11110000
int b = 170        // b = 10101010
```

& (bitwise Logical AND)

```
int x = (a & b);   // x = 10100000 (160)
```

| (bitwise Logical OR)

```
int x = (a | b);   // x = 11111010 (250)
```

^ (bitwise XOR)

```
int x = (a ^ b);   // x = 01011010 (90)
```

```
int c = 255        // c = 11111111
```

<< (bitwise Left Shift)

```
int x = (a << 3); // x = 11110000000 (1920)
int x = (a << 2); // x = 1111000000  (960)
```

>> (bitwise Right Shift)

```
int x = (a >> 3); // x = 00011111 (31)
int x = (a >> 2); // x = 00111111 (63)
int x = (c >> 3); // x = 00011111 (31)
int x = (c >> 2); // x = 00111111 (63)
```

Assignment operators

examples

=

```
int a = 100;   // a = 100
```

+=

```
a += 10;       // a = 110
```

-=

```
a -= 10;       // a = 90
```

*=

```
a *= 5;   // a = 500
```

/=

```
a /= 5;    // a = 20
```

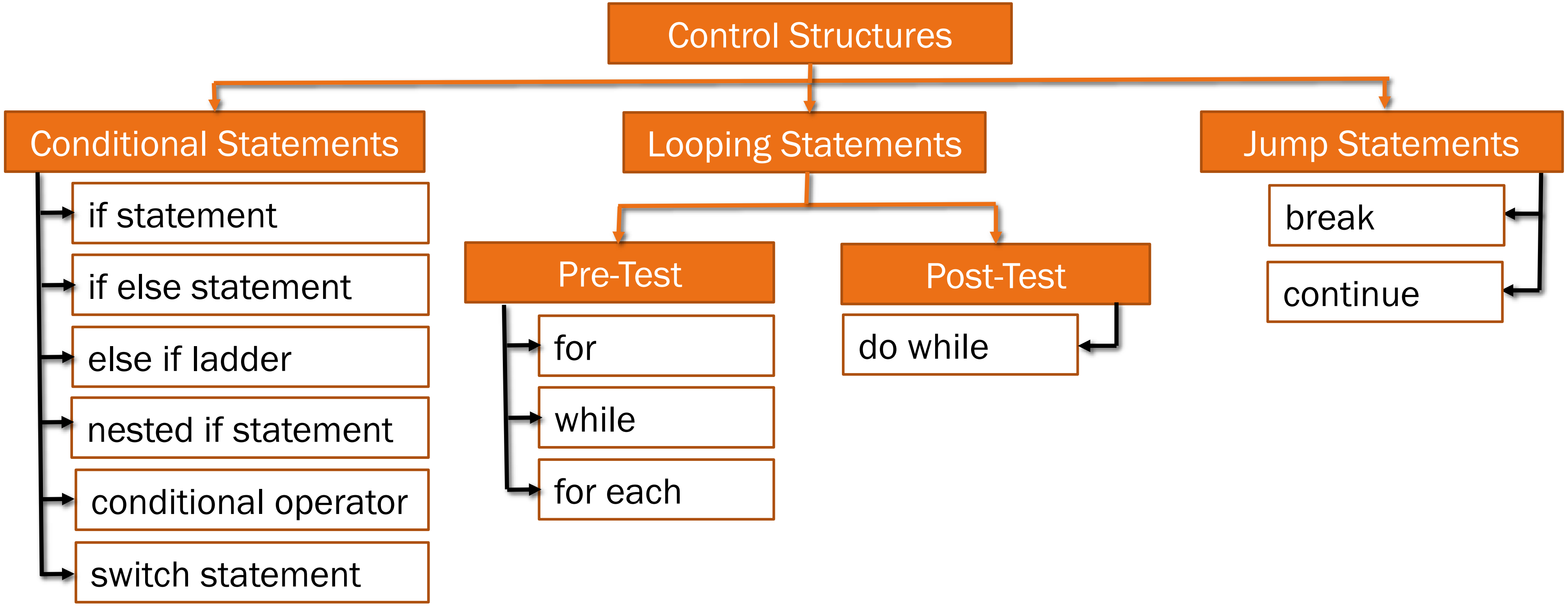
%=

```
a %= 3;    // a = 1
```

Control structures

- Control structures are used to control the steps of a program or instructions.
- Java provides **Control structures** that can change the path of execution and control the execution of instructions.

Control structures



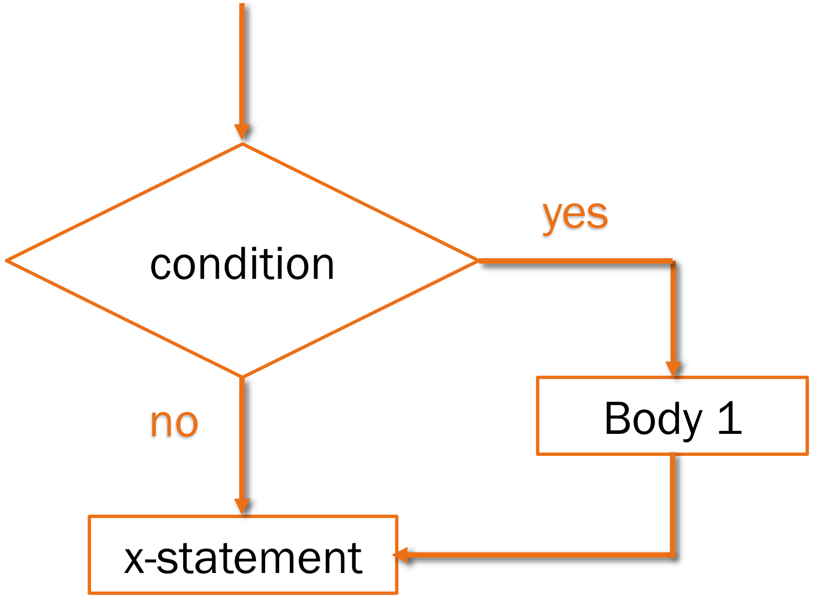
if Statement

syntax

Syntax

```
.................
if( condition ) {
     Body 1
     ..........
     ..........
}
x-statement;
```

Flowchart

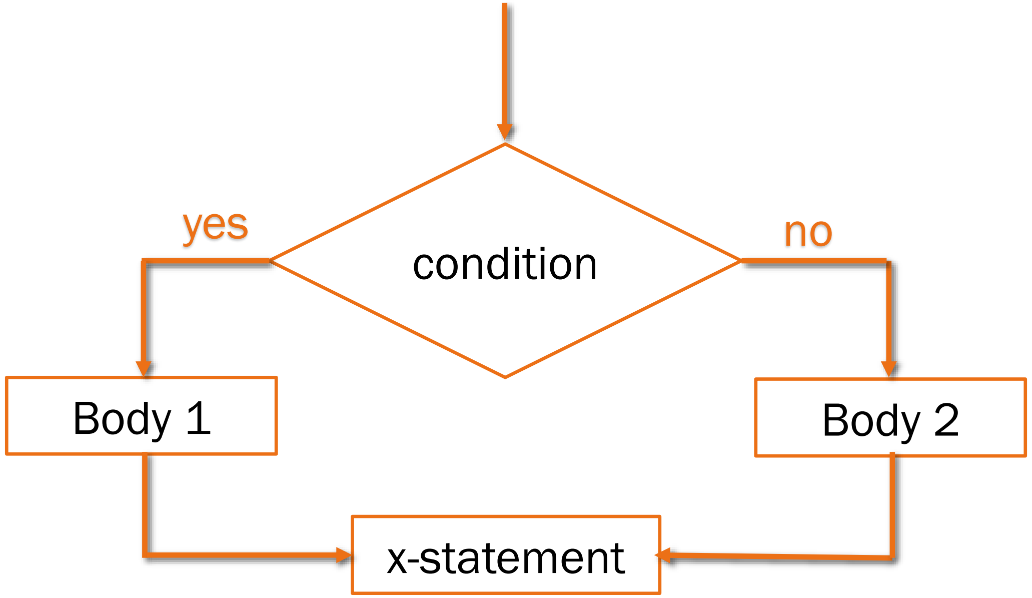


if-else Statement

Syntax

```
. . . . . . . . . . . . . . . . .
if( condition ) {
        Body 1
        . . . . . . . . . .
} else {
        Body 2
        . . . . . . . . . .
}
x-statement;
```

Flowchart



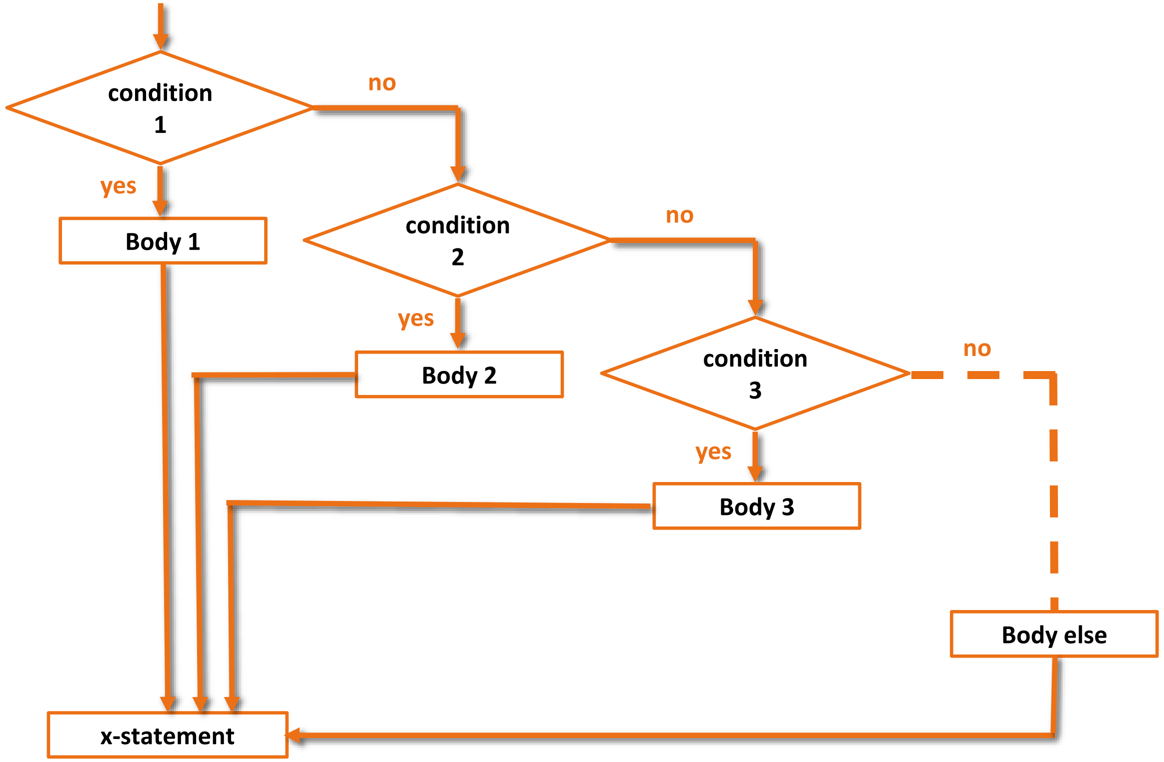
else-if ladder

syntax

Syntax

```
        ..................

        if( condition 1 ) {

                Body 1..........

        } else if( condition 2){

                Body 2..........

        }

.......................

        else {

                Body else..........

        }

        x-statement;
```

Flowchart

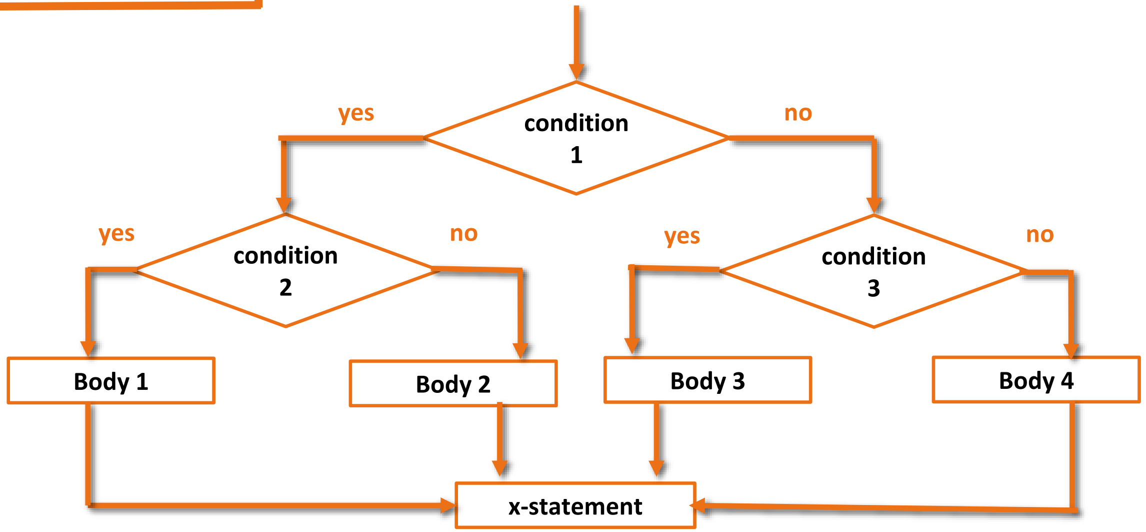


nested if

syntax

```
if( condition 1 ) {
        if( condition 2) {
                Body 1........
        } else {
                Body 2........
        }
} else {
        if( condition 3) {
                Body 3........
        } else {
                Body 4........
        }
}
x-statement;
```

Flowchart



Switch statement

syntax

```
switch( var ) {
       case c1 :
              ...
              break;
       case c2 :
              ...
              break;
       case c3 :
              ...
              ...
              break;
       ...
       ...
       ...
       default :
              ...
              break;
}
x-statement;
```

Questions

classwork

1) Write a program to take a number from user and print that number only if it is divisible by 10.

2) Write a program to take a number from user and print whether it is even or odd.

3) Write a program to take percentage from user and print the appropriate grade.

   0 – 39 → Fail, 40 – 44 → Pass class, 44 – 59 → Second class
   60 – 74 → First class, 75 – 100 → Distinction class, Otherwise → Invalid percentage

4) Write a program to take age and gender from user and print whether the person is eligible for marriage or not.

   a) Male and 21 and above eligible b) Female and 18 and above eligible c) Otherwise not eligible

5) Write a program to take day number of week and print the appropriate day.

   1 → Sunday, 2→ Monday, 3 → Tuesday, …….., 7 → Saturday, Otherwise → Invalid day

Questions

homework

1) Write a program to take two numbers from user and print the largest number.

2) Write a program to take three numbers from user and print the largest number.

3) Write a program to take a character from user and print whether it is vowel or consonant.

for Loop

syntax

```
for( initialization; condition; incr/decr ) {
        ...
        loop body
        ...
}
```

```
Q. Print “Hello world!” five times.
for( int i = 1; i <= 5; i++ ) {
        System.out.println(“Hello world!”);
}
```

while Loop

syntax

```
initialization;
while( condition ) {
          ...
          loop body
          incr/decr;
}
```

Q. Print “Hello world!” five times.

```
int i = 1;
while(i <= 5) {
          System.out.println(“Hello world!”);
          i++;
}
```

do while Loop

syntax

```
initialization;
do {
          ...
          loop body
          incr/decr;
} while( condition );
```

Q. Print “Hello world!” five times.

```
int i = 1;
do {
          System.out.println(“Hello world!”);
          i++;
} while(i <= 5);
```

for each Loop

syntax

```
for( var : list ) {

      ...

      loop body

      ...

}
```

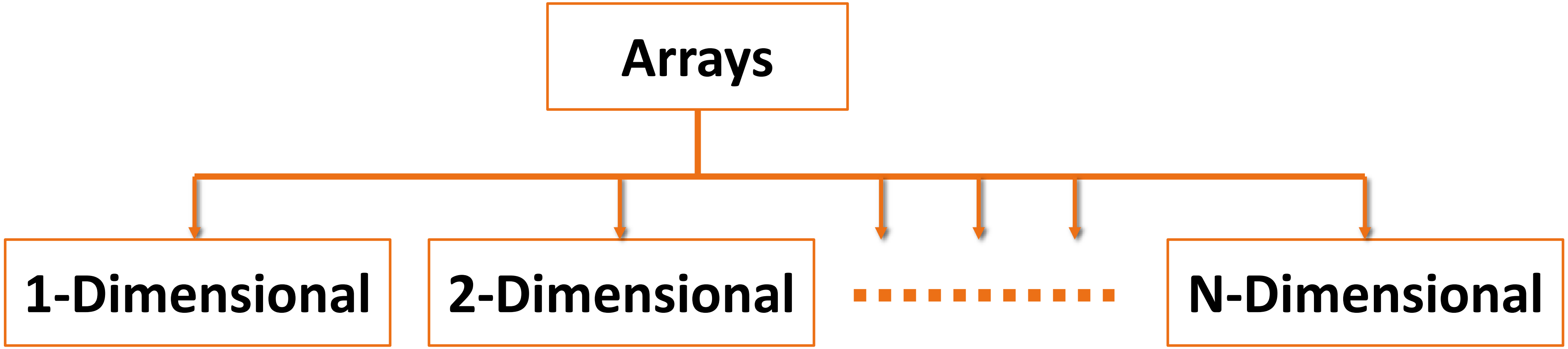
Q. Print the array having five items.

```
int arr[] = {10, 20, 30, 40, 50};

for( int x : arr ) {

      System.out.println(x);

}
```

Array

- Arrays are **finite set** of **homogeneous elements**.
- Elements of arrays are stored on continuous memory locations.
- To access these elements of array, we use indexes.
- In java index values can have byte, short and int as data type, but can’t have long.
- Index of an array starts from 0 and ends at (n-1), where n is the size of array.
- In java array is an object of dynamically generated class which inherits from Object class and implements Cloneable interface.

Types of Arrays



Arrays

1-Dimensional

2-Dimensional

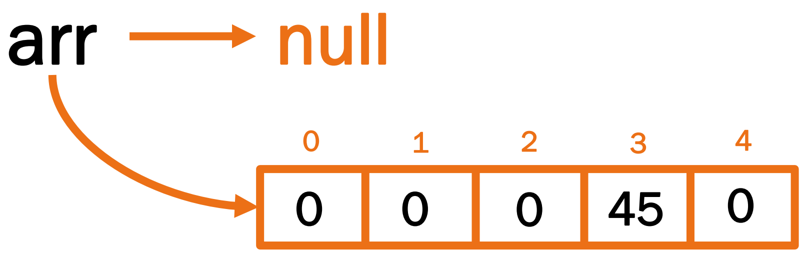
N-Dimensional

1-D Array

syntax

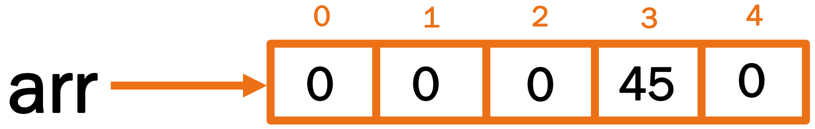
1st Way

```
int arr[];              // Declaration

arr = new int[5];       // Creation

arr[3] = 45;            // Initialization
```



2nd Way // Declaration & Creation

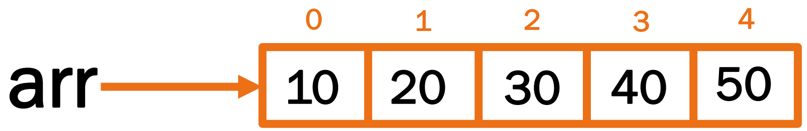
```
int arr[] = new int[5];

arr[3] = 45;            // Initialization
```



3rd Way

```
int arr[] = {10, 20, 30, 40, 50};
```

Declaration,Creation & Initialization

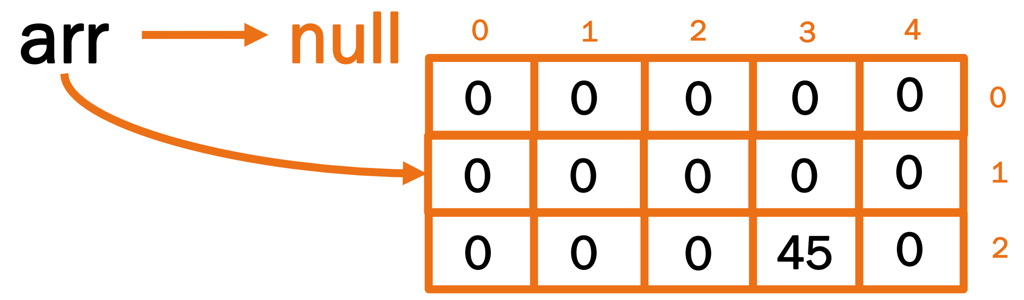


2-D Array

syntax

1st Way

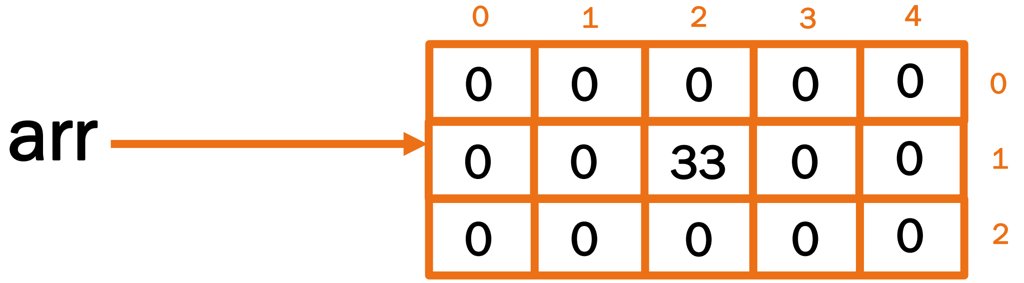
```
int arr[][];

arr = new int[3][5];

arr[2][3] = 45;
```



	0	1	2	3	4	
	0	0	0	0	0	0
	0	0	0	0	0	1
	0	0	0	45	0	2

2nd Way

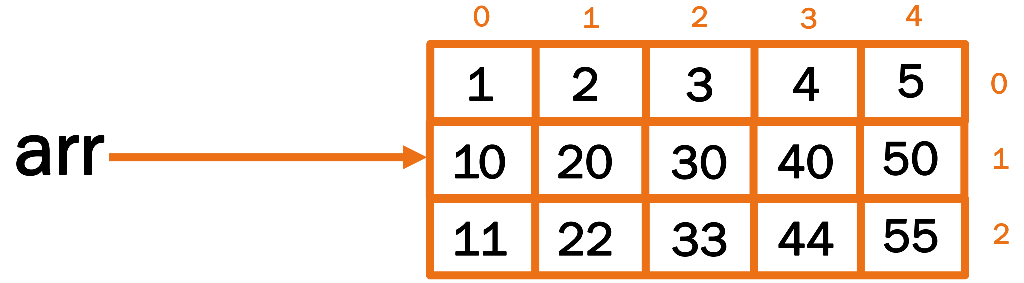
```
int arr[][] = new int[3][5];

arr[1][2] = 33;
```



	0	1	2	3	4	
	0	0	0	0	0	0
	0	0	33	0	0	1
	0	0	0	0	0	2

3rd Way

```
int arr[][] = {   {1, 2, 3, 4, 5},

                  {10, 20, 30, 40, 50},

                  {11, 22, 33, 44, 55}   };
```



	0	1	2	3	4	
	1	2	3	4	5	0
	10	20	30	40	50	1
	11	22	33	44	55	2

Methods

- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
- It is used to achieve the **reusability** of code.
- We write a method once and use it many times.
- It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code.
- The method is executed only when we call or invoke it.

Methods

syntax

```
access      access      return   method       (  argument list  )
specifier   modifier    type     identifier
{
        . . . . . . . . . .

        . . .    Body    . . .

        . . . . . . . . . .
}
```

Types of Methods

1. Method with no argument(s) no return type
2. Method with argument(s) but no return type
3. Method with no argument(s) but have return type
4. Method with argument(s) and with return type

Type 1

examples

```
import java.util.*;

class MiniCalcy {

       public void add() {

int a, b, c;

Scanner sc = new Scanner(System.in);

a = sc.nextInt();

b = sc.nextInt();

c = a + b;

System.out.println(“Addition :” + c);

        }

}
```

```
import java.util.*;

class Demo {

public static void main(String args[]) {

       MiniCalcy obj = new MiniCalcy();

       obj.add();

        }

}
```

Type 2

examples

```
class MiniCalcy {

public void add(int a, int b) {

       int c;

       c = a + b;

System.out.println(“Addition :” + c);

       }

}
```

```
import java.util.*;

class Demo {

public static void main(String args[]) {

    MiniCalcy obj = new MiniCalcy();

    Scanner sc = new Scanner(System.in);

    x = sc.nextInt();

    y = sc.nextInt();

    obj.add(x, y);

        }

}
```

Type 3

examples

```
class MiniCalcy {

public int add() {

    Scanner sc = new Scanner(System.in);

    int a = sc.nextInt();

    int b = sc.nextInt();

    int c = a + b;

    return c;

        }

}
```

```
import java.util.*;

class Demo {

public static void main(String args[])
{

    MiniCalcy obj = new MiniCalcy();

    int z = obj.add();

System.out.println(“Addition :” + z)

        }

}
```

Type 4

examples

```
class MiniCalcy {

public int add(int a, int b) {

               int c;

               c = a + b;

               return c;

        }

}
```

```
import java.util.*;

class Demo {

public static void main(String args[]) {

    MiniCalcy obj = new MiniCalcy();

    Scanner sc = new Scanner(System.in);

    int x = sc.nextInt();

    int y = sc.nextInt();

    int z = obj.add(x, y);

    System.out.println(“Addition :” + z);

        }

}
```

Pillars of OOPs

1) Abstraction

2) Encapsulation

3) Polymorphism

- Static Polymorphism
- Dynamic Polymorphism

4) Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance

Encapsulation

- Wrapping up of data and functions in a wrapper know as class is called as encapsulation.
- As Java is a pure Object Oriented Language, Encapsulation is must in Java.
- All the variables in the class are called as **data members** and all the methods in the class are **called** as **member functions**, collectively they are called as **class members**.
- To access these members, we either require object of that class if they are non-static or name of that class if they are static.
- Encapsulation automatically achieve the concept of data hiding providing security to data by making the variable as private and expose the property to access the private data which would be public.

Class & Objects

- Classes and objects are the basic building blocks in Object Oriented Programming.
- A class is a user defined template or blueprint or prototype from which objects are created.
- For creating a named class, we use **class** keyword followed by class identifier.
- Object is a **run-time entity;** object represents the real world.
- For creating an object, we use **new** keyword.
- Objects have state(attributes) and behaviour(methods).
- Constructor gets executed whenever we create object of a class.

Class

syntax

access specifier access modifier class class identifier extends CN implements IN

{

. . . .

. . Body . . .

. . . .

}

Class

example

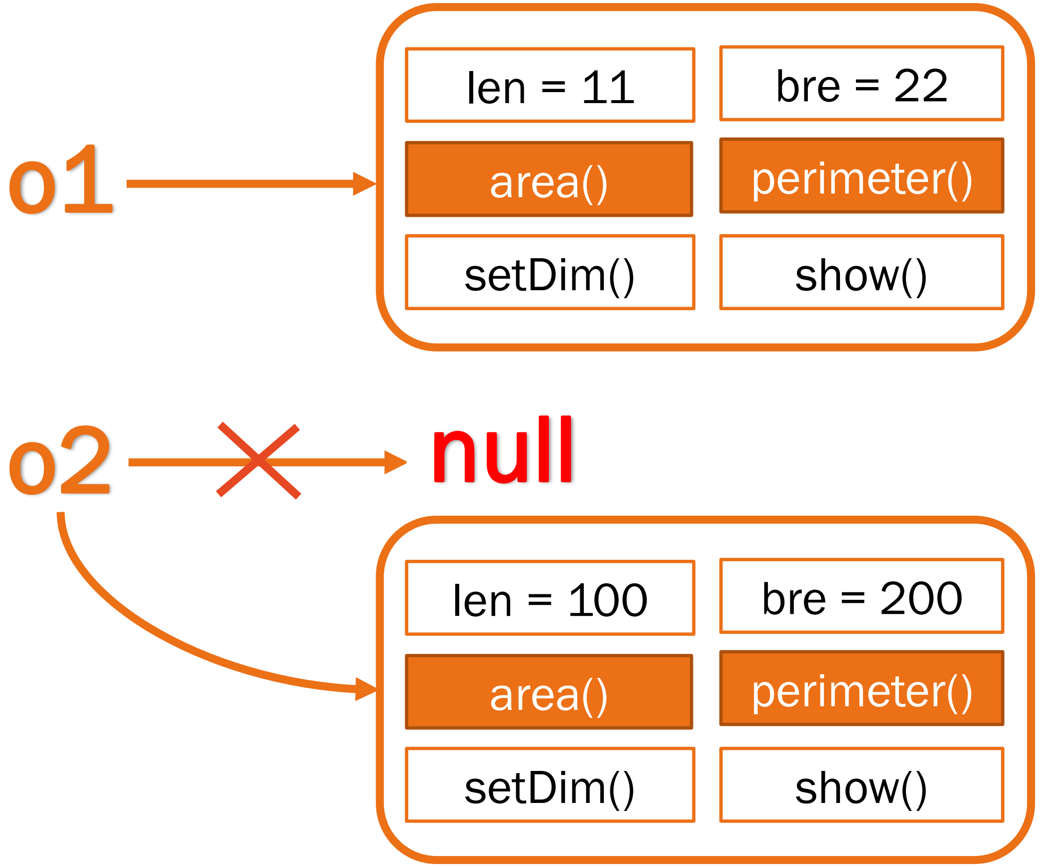
```
class Rect {
  public int len, bre;
  public void setDim(int l, int b) {
        len = l;
        bre = b;
  }
  private int area (){
        int a = len * bre;
        return a;
  }
```

```
private int perimeter() {
        int p = 2 * (len + bre);
        return p;
 }
 public void show (){
  System.out.println(“ Len : ” + len);
  System.out.println(“ Bre : ” + bre);
  System.out.println(“ Area : ” + area());
  System.out.println(“ Perimeter : ” + perimeter());
 }
}
```

Object

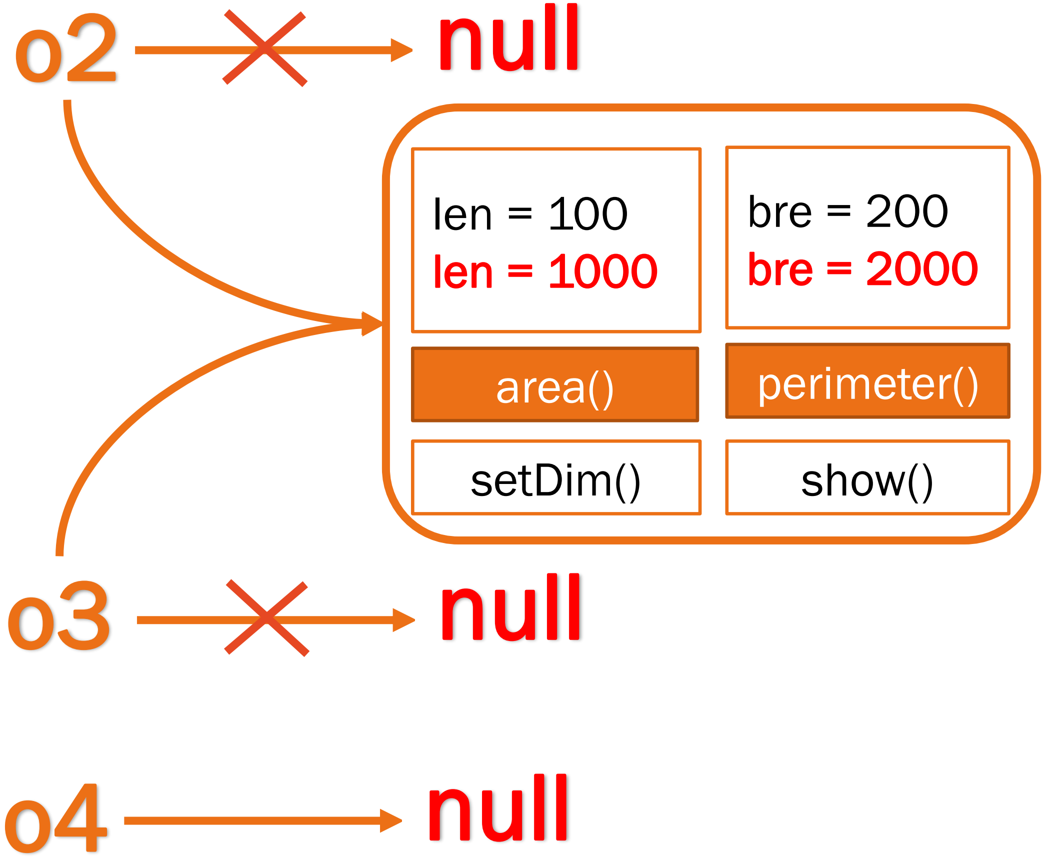
example

```
class Demo {

  public static void main(String args[]) {

        Rect o1 = new Rect();

        o1.len = 11;

        o1.bre = 22;

        o1.show();


        Rect o2;

        o2 = new Rect();

        o2.setDim(100, 200);

        o2.show();
```



Object cont... example

```
        Rect o3;

        o3 = o2;

        o3.show ();


        o3.setDim(1000, 2000);

        o1.show();

        o2.show();

        o3.show();

        Rect o4;

        o4.setDime(1,1);

    }

}
```



Constructor

- Constructor is a special type of method.
- Constructor have same name as class name.
- Constructor don’t have any return type, not even void.
- Constructor gets executed whenever we create object of that class.
- Constructors are generally used to initialize instance variables.
- Constructor who don’t take any arguments are called as **default constructor**.
- Constructor who take arguments are called as **parametrized constructor**.
- Constructor who take argument of type itself is called as **copy constructor**.

Constructor cont…

- Private constructors are used to restrict the object creation of a class from other classes.
- Every class should have at least one constructor in it.
- If we don’t provide any constructor, then compiler automatically puts one default constructor in our class.
- First line in any constructor should be a call to its super class constructor.
- If we don’t provide this call then compiler automatically puts a call to the default constructor of the super class.

Constructor

syntax

access specifier   class-name/ constructor-identifier (argument list)

{

. . . .

. .   Body   . . .

. . . .

}

Constructor

example

```
class Rect {

   public int len, bre;

   public Rect() {

         len = bre = 0;

   }

   public Rect(int x) {

         len = bre = x;

   }

   public Rect(int l, int b) {

         len = l;

         bre = b;

   }
```

```
public Rect(Rect obj) {

        this.len = obj.len;

       this.bre = obj.bre;

}

public void show (){

 System.out.println(“ Len : ” + len);

 System.out.println(“ Bre : ” + bre);

 System.out.println(“ Area : ” + area());

 System.out.println(“ Perimeter : ” + perimeter());

}
```

Constructor

example

```
  private int area (){
          int a = len * bre;
          return a;
  }
  private int perimeter() {
          int p = 2 * (len + bre);
          return p;
  }
}
```

```
class Demo {
   public static void main(String  args[])
   {
         Rect o1 = new Rect(10,20);
         o1.show();

         Rect o2 = new Rect(10);
         o2.show();

         Rect o3 = new Rect();
         o3.show();
```

```
      Rect o4 = o2
      o4.show();

      Rect o5 = new Rect(o1);
      o5.show();
    }
}
```

Polymorphism

- Polymorphism means one thing taking many forms.
- It is derived from two Greek words; **poly** means **many** and **morphs** means **forms**.
- There are two types of polymorphism in java.
- **Static polymorphism :** It is also called as **early binding** or **compile time polymorphism.**
- This type of polymorphism is implemented with the help of **overloading.**
- **Dynamic polymorphism :** It is also called as **late binding or run time polymorphism**.
- This type of polymorphism is implemented with the help of **overriding.**

Overloading

- Method with same name but different argument list is called as method overloading.
- Based on the type of parameters passed, method is selected for invocation.
- Based on the parameters passed, the call gets bind with the method at the time of compilation, that’s why it is called as compile time polymorphism or early binding or static polymorphism.
- Both methods and constructors can be overloaded.

Overriding

- Method with same name and same argument list but one in super class and another in sub class is called as method overriding.
- Based on the object used for calling, method is selected for invocation.
- If we use object of super class for invocation, then super class method is invoked; if we use object of sub class for invocation, then sub class method is invoked.
- Based on the object, the call gets bind with the method, as object is a runtime entity, this binding is done at runtime, that's why it is called as run time polymorphism or late binding or dynamic polymorphism.
- We can only override methods; we cannot override constructors.
- We cannot override static methods.

Inheritance

- Inheritance is the ability of one object to acquire some/all properties of another object.
- In biological inheritance a child inherits the traits of his/her parents, in the same way child class inherits some or all members of parent class.
- Major advantage of inheritance is **code reusability.**
- You can reuse the fields and methods of the existing class and add the required fields and methods as per need in child class.
- The class whose properties are inherited is called as **super class/parent class/base class.**
- The class who inherits the properties is called as **sub class/child class/derived class**.
- Conceptually there are four types of inheritance, **Single Inheritance, Multi-level Inheritance, Hierarchical Inheritance** and **Multiple Inheritance**.
- For inheriting class, we use **extends** keyword and for inheriting from Interface we use **implements** keyword.

Inheritance

syntax

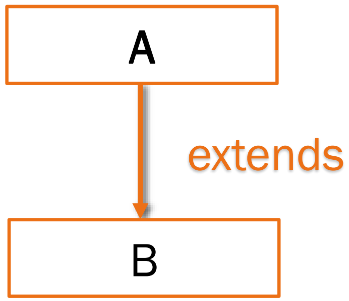
A super class reference variable can point to a sub class object but can access only those members in sub class which are inherited from super class.

Single Inheritance

Syntax

```
class A {
       ..........
       ..........
}
class B extends A {
       ..........
       ..........
}
```

Flowchart



Single Inheritance

example

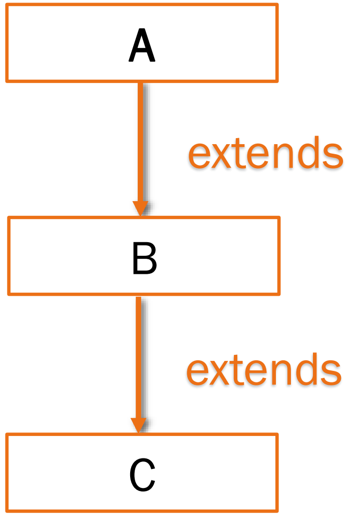
```
class A {
   public void m1() {
      System.out.println(“class A : m1“);
   }
   public void m2() {
      System.out.println(“class A : m2“);
   }
}
class B extends A {
   public void m1() {
      System.out.println(“class B : m1“);
   }
   public void m3() {
      System.out.println(“class B : m3“);
   }
}
class Demo {
   public static void main(String args[]) {
         A o1 = new A ();
         o1.m1();
         o1.m2();
          o1.m3(); X
         B o2 = new B ();
              o2.m1();
              o2.m2();
               o2.m3();
         A o3 = new B ();
              o3.m1();
              o3.m2();
               o3.m3(); X
   }
}
```

Multilevel Inheritance

Syntax

```
class A {
        ..........
}
class B extends A {
        ..........
}
class C extends B {
        ..........
}
```

Flowchart



Multi-level Inheritance

example

```
class A {
    public void m1() {
        System.out.println(“class A : m1”);
    }
    public void m2() {
        System.out.println(“class A : m2”);
    }
}
class B extends A {
    public void m1() {
        System.out.println(“class B : m1”);
    }
    public void m3() {
        System.out.println(“class B : m3”);
    }
    public void m4() {
        System.out.println(“class B : m4”);
    }
}
class C extends B {
    public void m1() {
        System.out.println(“class C : m1”);
    }
```

Multi-level Inheritance

example

```
   public void m3() {
      System.out.println(“class C : m3“);
   }
   public void m5() {
      System.out.println(“class C : m5“);
   }
}
class Demo {
   public static void main(String args[]) {
         A o1 = new A();
         o1.m1();
         o1.m2();
```

```
         B o2 = new B();
         o2.m1();
         o2.m2();
         o2.m3();
         o2.m4();
         C o3 = new C();
         o3.m1();
         o3.m2();
         o3.m3();
         o3.m4();
         o3.m5();
         A o4 = new B();
```

```
         o4.m1();
         o4.m2();
         A o5 = new C();
         o5.m1();
         o5.m2();
         B o6 = new C ();
         o6.m1();
         o6.m2();
         o6.m3();
         o6.m4();
   }
}
```

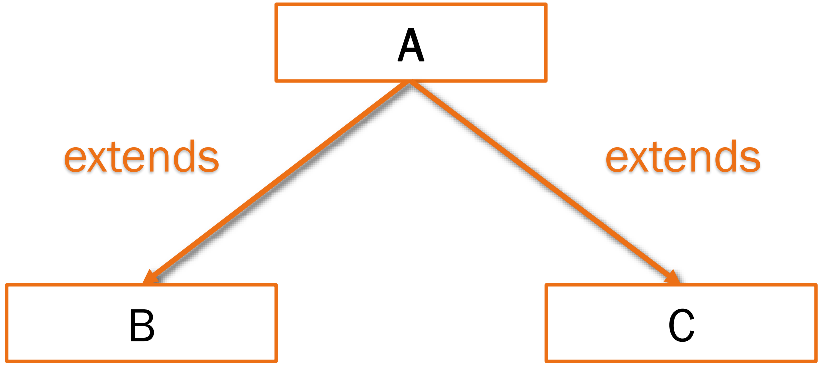
Hierarchical Inheritance

syntax

Syntax

```
class A {
        ..........
}
class B extends A {
        ..........
}
class C extends A {
        ..........
}
```

Flowchart



Hierarchical Inheritance

example

```
class A {
   public void m1() {
      System.out.println(“class A : m1“);
   }
   public void m2() {
      System.out.println(“class A : m2“);
   }
}
class B extends A {
   public void m1() {
      System.out.println(“class B : m1“);
   }
   public void m3() {
      System.out.println(“class B : m3“);
   }
}
class C extends A {
   public void m1() {
      System.out.println(“class C : m1“);
   }
   public void m4() {
      System.out.println(“class C : m4“);
   }
}
```

Hierarchical Inheritance

example

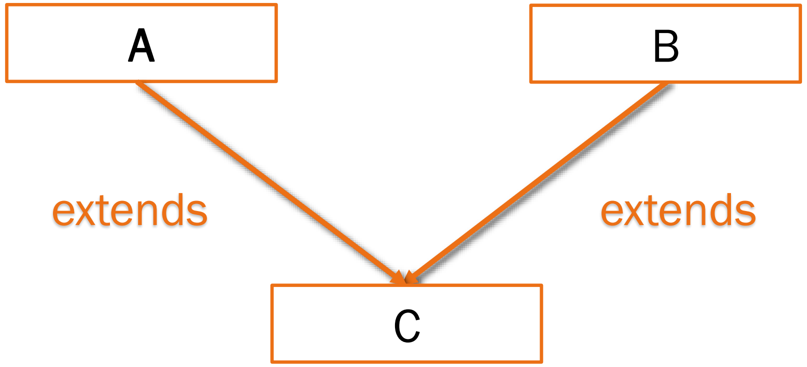
```
class Demo {

   public static void main(String args[]) {

         A o1 = new A();

         o1.m1();

         o1.m2();

         B o2 = new B();

         o2.m1();

         o2.m2();

         o2.m3();

         C o3 = new C();

         o3.m1();

         o3.m2();

         o3.m4();


         A o4 = new B();

         o4.m1();

         o4.m2();


         A o5 = new C();

         o5.m1();

         o5.m2();

     }

 }
```

Multiple Inheritance **Not supported in java** **syntax**

Syntax

```
class A {
        ..........
}
class B {

        ..........

}
class C extends A, B {

        ..........

}
```

Flowchart



Hybrid Inheritance

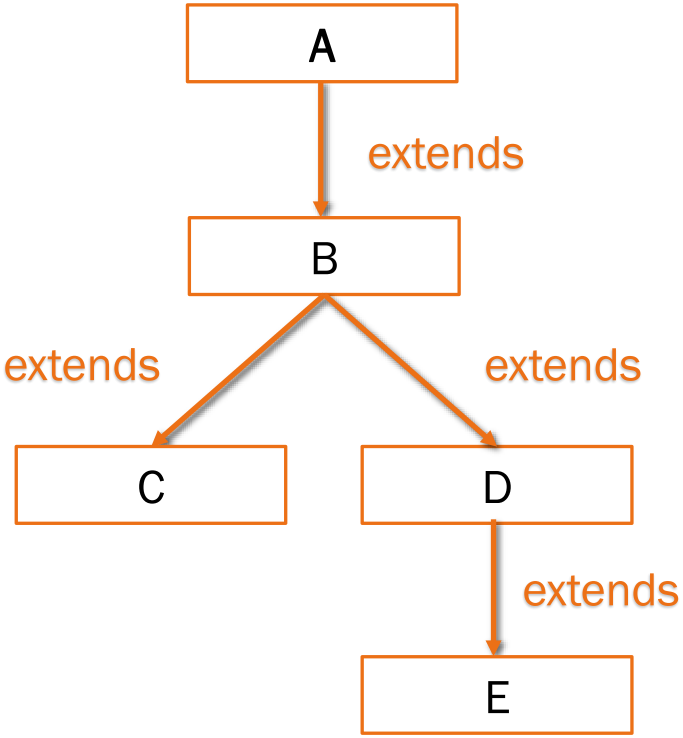
syntax

Syntax

```
class A {
       ..........
}
class B extends A {
       ..........
}
class C extends B {
       ..........
}
```

```
class D extends B {
       ..........
       ..........
}
class E extends D {
       ..........
       ..........
}
```

Flowchart



Polymorphism

- Polymorphism means one thing taking many forms.
- It is derived from two Greek words; **poly** means **many** and **morphs** means **forms**.
- There are two types of polymorphism in java.
- **Static polymorphism :** It is also called as **early binding** or **compile time polymorphism.**
- This type of polymorphism is implemented with the help of **overloading.**
- **Dynamic polymorphism :** It is also called as **late binding or run time polymorphism**.
- This type of polymorphism is implemented with the help of **overriding.**

Overloading

- Method with same name but different argument list is called as method overloading.
- Based on the type of parameters passed, method is selected for invocation.
- Based on the parameters passed, the call gets bind with the method at the time of compilation, that’s why it is called as compile time polymorphism or early binding or static polymorphism.
- Both methods and constructors can be overloaded.

Overloading

example

```
class Addition {
   public void add(int a, int b) {
      System.out.println(“Sum : ” + (a+b));
   }
   public void add(double a, double b) {
      System.out.println(“Sum : ” + (a+b));
   }
   public void add(int lst[]) {
      int sum = 0;
      for(int x : lst) {
          sum += x;
       }
      System.out.println(“Sum : ” + sum);
    }
}
class Demo {
   public static void main(String args[]) {
         Addition obj = new Addition();
         obj.add(10.45, 20.85);
         obj.add(100, 200);
         int arr[] = {10,20,30,40,50};
         obj.add(arr);
   }
}
```

Overriding

- Method with same name and same argument list but one in super class and another in sub class is called as method overriding.
- Based on the object used for calling, method is selected for invocation.
- If we use object of super class for invocation, then super class method is invoked; if we use object of sub class for invocation, then sub class method is invoked.
- Based on the object, the call gets bind with the method, as object is a runtime entity, this binding is done at runtime, that’s why it is called as run time polymorphism or late binding or dynamic polymorphism.
- We can only override methods; we cannot override constructors.
- We cannot override static methods.

Overriding

example

```
class A {
   public void m1() {
      System.out.println(“class A : m1“);
   }
   public void m2() {
      System.out.println(“class A : m2“);
   }
}
class B extends A {
   public void m1() {
      System.out.println(“class B : m1“);
   }
   public void m3() {
      System.out.println(“class B : m3“);
   }
}
class Demo {
   public static void main(String args[]) {
         A o1 = new A ();
         o1.m1();
         o1.m2();
          o1.m3();
         B o2 = new B ();
               o2.m1();
               o2.m2();
                o2.m3();
         A o3 = new B ();
               o3.m1();
               o3.m2();
                o3.m3();
     }
}
```

this keyword

- this keyword means object of same class or current class.
- this keyword is used to access the instance members of current class.

```
class Rect {
      int len, bre;
      public Rect(int len, int bre)
      {
            this.len = len;
            this.bre = bre;
      }
      public int area() {
            int a = this.len * this.bre;
            return a;
      }
      public void show() {
            int op = this.area();
            System.out.println(“Area : ‟ + op);
      }
}
```

super

keyword

- super keyword when used with dot operator it acts as an object of super class.
- super keyword when used with parenthesis then it is use to invoke super class constructor.

```
class Rect {
        int len, bre;
        public Rect(int len, int bre) {
                this.len = len;
                this.bre = bre;
        }
}
class Box extends Rect {
        int hei;
```

```
        public Box(int l, int b, int h) {
        super(l, b);
        this.hei = h;
    }
    public void show(){
        System.out.println(super.len + “ “
                super.bre + “ “ + this.hei);
    }
}
```

final keyword

- When we use final keyword with a variable then we cannot reinitialize that variable.

```
final int a = 10;

a = 35; error

a = 90; error
```

- When we use final keyword with an object then we cannot re-instantiate that object.

```
final Rect o1 = new Rect(10,20);

o1 = new Rect(11,22); error

o1 = new Rect(0, 0);  error
```

- When we use final keyword with a method then we cannot override that method.

```
class A {

   public final void m1(){

   }

}
```

```
class B extends A{

    public void m1(){

    }

}
```

- When we use final keyword with a class then we cannot inherit from that class.

```
final class A {

   public void m1(){

   }

}
```

```
class B extends A{

    public void m1(){

    }

}
```

Package

example

```
package campuscorners.pkg1;
public class A {
        ..........
}
```

```
package campuscorners.pkg1;
public class B {
        ..........
}
```

```
package campuscorners.pkg1;
public class C {
        ..........
}
```

```
package campuscorners.pkg2;
public class D {
        ..........
}
```

```
package campuscorners.pkg3;
public class E {
        ..........
}
```

```
package campuscorners.demo;
class E {
        ..........
}
```

```
package campuscorners.demo;
import campuscorners.pkg1.*;
import campuscorners.pkg2.D;
class Demo {
   public static void main(String  args[]){
         A o1 = new A();
         B o2 = new B();
         C o3 = new C();
         D o4 = new D();
campuscorners.pkg3.E o5 = new campuscorners.pkg3.E();
         E o5 = new E();
   } }
```

Package

… compile & run

Compile

```
javac –d . FileName.java
```

OR

```
javac –d . *.java
```

Run

```
java pkg.subpkg.ClassName
```

Abstraction

- Abstraction is also called as **details hiding**.
- Abstraction is the process of showing only essential/necessary features of an entity/object to the outside world and hide the other irrelevant information.
- To achieve abstraction we can use abstract keyword.
- Complete abstraction can be achieved with the help of interfaces.

abstract keyword

- **With method:**

  i) When we use abstract keyword with method then we don’t provide body or implementation for that method.

  ii) It means abstract methods don’t have any implementation.

  iii) Implementation to these methods are given in subclasses by overriding these methods.

  iv) A class containing an abstract method should be declared as abstract.

```
abstract class Arithmetic {
    public int add(int a, int b) {
        return (a + b);
    }
    public int sub(int a, int b) {
        return (a - b);
    }
    public int mul(int a, int b) {
        return (a * b);
    }
    public abstract int div(int a, int b);
}
```

abstract cont… keyword

- **With class:**

  i) Whenever a class contains an abstract it must be declared as abstract.

  ii) An abstract class may contain all abstract methods or all concrete methods or combination of both.

  iii) We cannot create object of an abstract class, but we can create reference variable of it.

  iv) Any class inheriting from an abstract class should give body to all the abstract methods of its super class and override it.

  v) If this sub-class don't give body to even for a single abstract method, then this class should also be declared as abstract.

```
abstract class Arithmetic {

    public abstract int div(int a, int b);

}

class Maths extends Arithmetic {

    public int div(int a, int b) {

        return (a/b);

    }

}
```

```
Arithmetic o1;

Arithmetic o2 = new Arithmetic();

Arithmetic o3 = new Maths();
```

Interface

- Interface is same as class.
- All the methods in an interface are by default public and abstract.
- All the variables in an interface are by default public, static and final.
- We cannot create object of an interface, but we can create reference variable of it.
- A class can inherit from an interface with the help of implements keyword.
- A class can inherit from multiple interfaces, i.e. supports multiple inheritance.
- An interface can inherit from another interface with the help of extends keyword.
- An interface can inherit from multiple interfaces, i.e. supports multiple inheritance.

Interface

syntax

access specifier interface interface identifier extends Interface Name

{

. . . .

. . Body . . .

. . . .

}

Interface

example

```
interface Animal {
    public abstract void talk();
}
class Cat implements Animal {
   public void talk(){
       SOP("Mewooooo...");
   }
}
class Dog implements Animal {
   public void talk(){
        SOP("Bhooooo...");
    }
}
class Cobra implements Animal {
   public void talk(){
        SOP("Hisssss...");
   }
}
class Demo {
   p.. s.. v.. m..(S.. A[]) {
       Animal obj = new Animal();
       obj.talk();
       Animal d = new Dog();
       d.talk();
       Animal c = new Cobra();
       c.talk();
       Animal t = new Cat();
       t.talk();
   }
}
```

instanceof keyword

instanceof keyword in used for comparing an object or instance with class name.

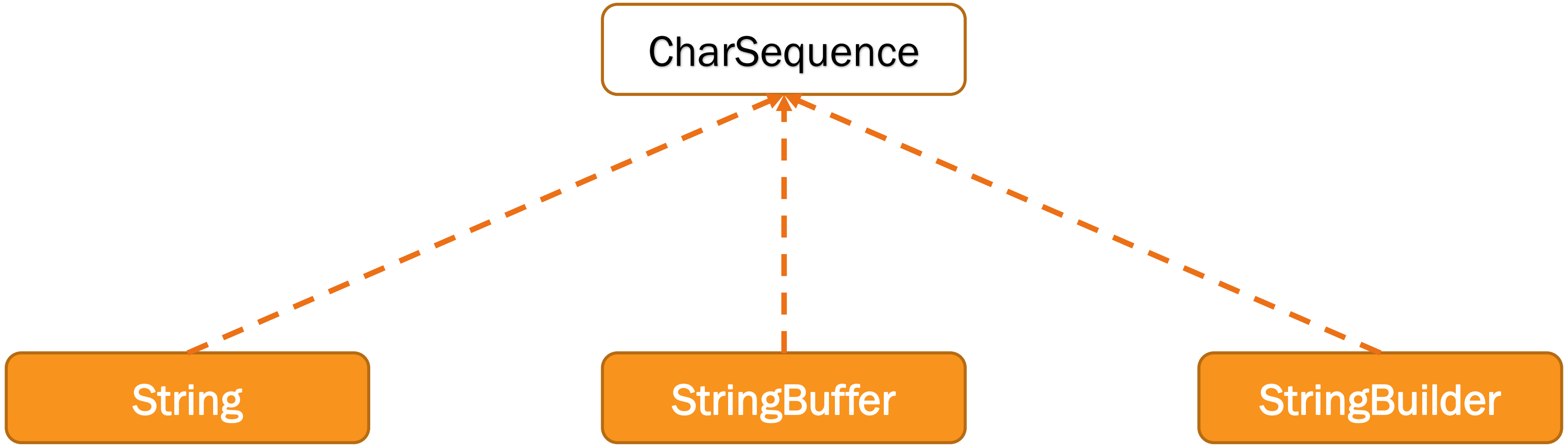
```
interface Animal {
}
class Cat implements Animal {
}
class Dog implements Animal {
}
class Cobra implements Animal {
}
class Demo {
  public static void show(Animal o){
        if(o instanceof Cat) {
           SOP(“It’s a cat...”);
        } else if(o instanceof Dog) {
           SOP(“It’s a dog...”);
        } else if(o instanceof Cobra) {
           SOP(“It’s a Cobra...”);
        }
  }
  p.. s.. v.. m..(S.. A[]) {
        Animal obj;
          obj = new Dog();
          show(obj);

          obj = new Cobra();
          show(obj);

          obj = new Cat();
          show(obj);
     }
}
```

Strings

- String is a class in Java which is used to store character sequences.
- String class is a final class in java.lang package, who inherits from **Comparable**, **Serializable** & **CharSequence** interfaces.
- String created with the help of String class is immutable, it means we cannot change the value of a String object.

String

syntax



Immutable String Objects

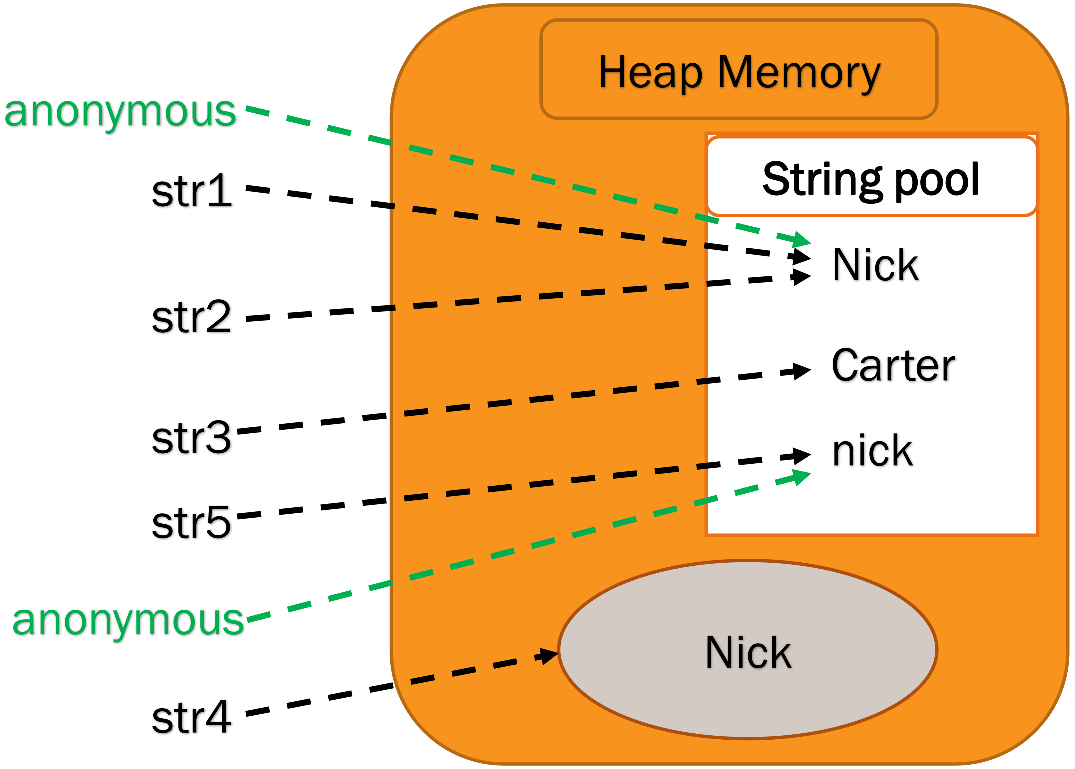
Mutable String Objects
Thread Safe

Mutable String Objects
Not Thread Safe

String

syntax

```
String str1 = "Nick";

String str2 = "Nick";

String str3 = "Carter";

String str4 = new String("Nick");

String str5 = "nick";

str1 == "Nick"
```



Wrapper Classes

- The **wrapper class in Java** provides the mechanism *to convert primitive into object and object into primitive*.
- Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Wrapper Classes cont...

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Wrapper Classes

cont…

- The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

- Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper Classes

cont…

Sr. No.	Primitive	Wrapper class
1	byte	java.lang.Byte
2	short	java.lang.Short
3	int	java.lang.Integer
4	long	java.lang.Long
5	float	java.lang.Float
6	double	java.lang.Double
7	char	java.lang.Character
8	boolean	java.lang.Boolean

Generics

- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

- You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Generics

syntax rules

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Collections Framework

- Framework is a set of classes and interfaces which provide a ready-made architecture to perform a particular task.

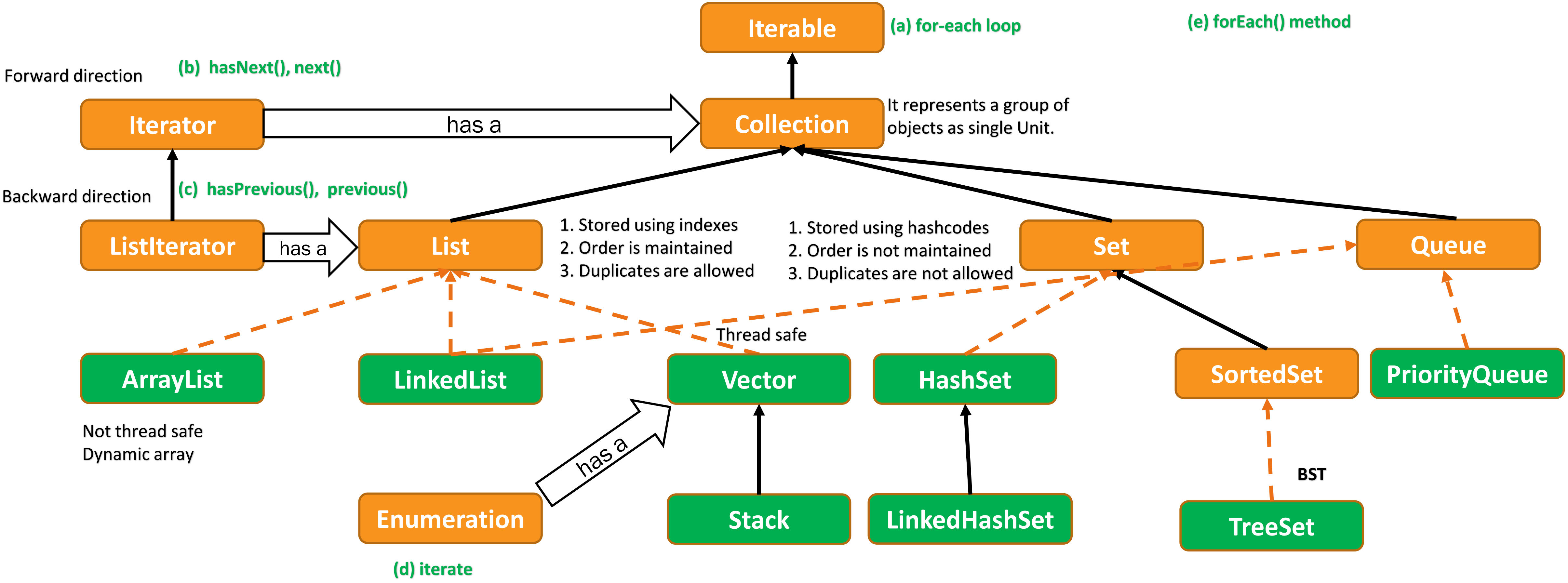
- Collection Framework is java API which provides architecture to store and manipulate group of objects.

- Any group of individual objects which are represented as a single unit is known as the collection of the objects.

- The **Collection** interface (java.util.Collection) and **Map** interface (java.util.Map) are the two main “root” interfaces of Java collection classes.

Collection Framework

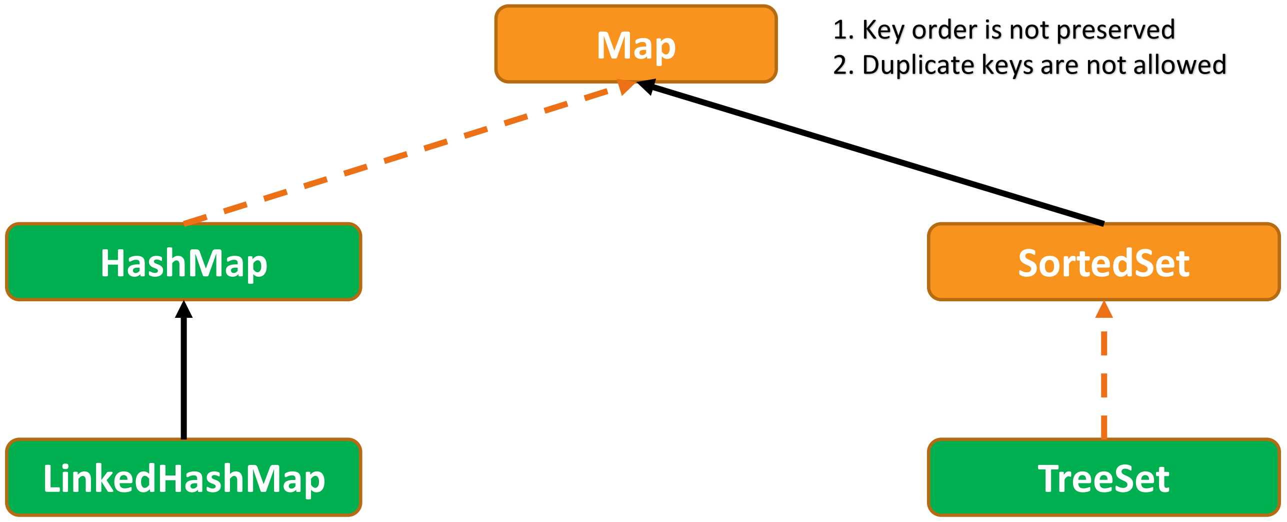
framework



Collection Framework

framework

key	value
name	Nikhil
age	34
height	5.4



1. Key order is not preserved
2. Duplicate keys are not allowed

1. Key order preserved
2. Duplicate keys are not allowed

1. Keywise sorting

Exception Handling

- An exception(error) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted, and the program/application terminates abnormally.
- The **Exception Handling** in Java is one of the powerful mechanism to **handle** the **runtime errors** so that normal flow of the application can be maintained.
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
- In Java runtime errors are categorised in 3 types
    1) Error
    2) Checked Exception
    3) Unchecked Exception

Exception Handling framework

- **Checked Exception:**

  The classes which directly inherit Exception class except RuntimeException and Error are known as checked exceptions.

  e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

- **Unchecked Exception:**

  The classes which inherit RuntimeException are known as unchecked exceptions.

  e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

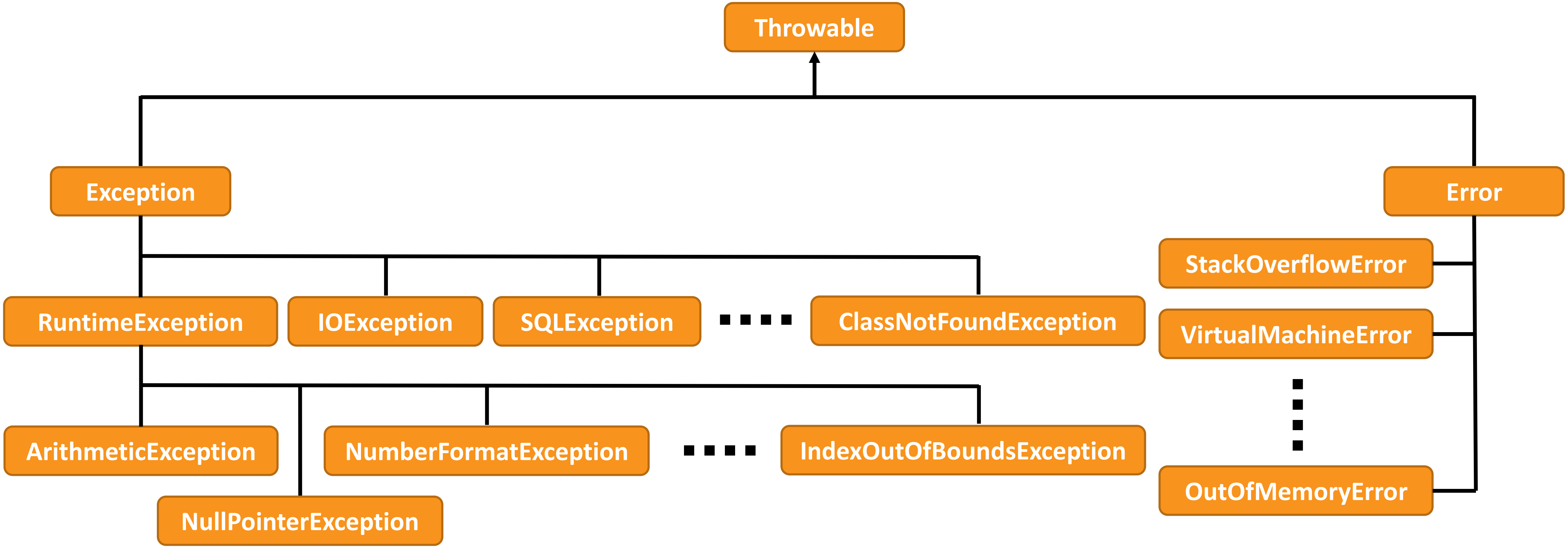
- **Error:**

  Error is irrecoverable. These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.

  e.g. OutOfMemoryError, VirtualMachineError, etc.

Exception Handling

hierarchy



Exception Handling

keywords

1) try
2) catch
3) finally
4) throw
5) throws

Exception Handling

syntax

1

```
try {
        .....
} catch(Exception e) {
        .....
}
x-statements
```

3

```
try {
        .....
} finally {
        .....
}
x-statemens
```

2

```
try {
        .....
} catch(Exe1 e) {
        .....
} catch(Exe2 e) {
        .....
}
.......
catch(Exception e) {
        .....
}
x-statements
```

4

```
try {
        .....
} catch(Exe1 e) {
        .....
}
.......
catch(Exception e) {
        .....
} finally {
        .....
}
x-statements
```

Exception Handling

syntax

5

```
try {
        .....
        try {

        } catch(Exception e) {
                .....
        }
        x-statements-1
} catch(Exception e) {
        ...
}
x-statements-2
```

6

```
public void div(int p1, int p2) throws Exe1, Exe2,..., ExeN {

}
```

7

```
class CustomExeception extends Exception {
        public CustomExeception() {
                ........
        }
}
```

```
CustomExeception err = new CustomExeception();
throw err;
```

File Handling

- Java **I/O (Input and Output)** is used to process the input and produce the output.
- Java uses the concept of a **stream** to make I/O operation fast.
- The **java.io** package contains all the classes required for input and output operations.
- We can perform file handling in Java by Java **I/O API**.
- A stream is a sequence of data. In Java, a stream is composed of bytes.
- **OutputStream:** Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- **InputStream:** Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

File Handling

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

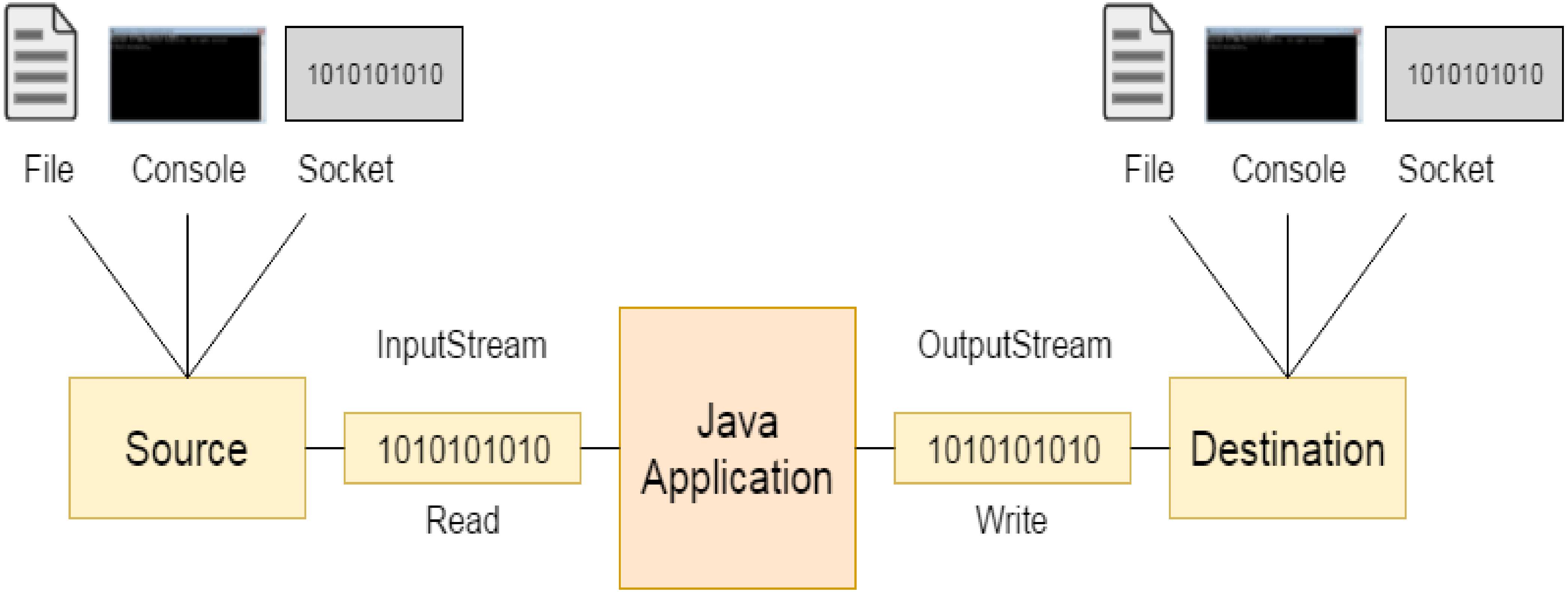
2) System.in: standard input stream

3) System.err: standard error stream

```
System.out.println("simple message");

System.err.println("error message");

int i=System.in.read();

System.out.println((char)i);
```

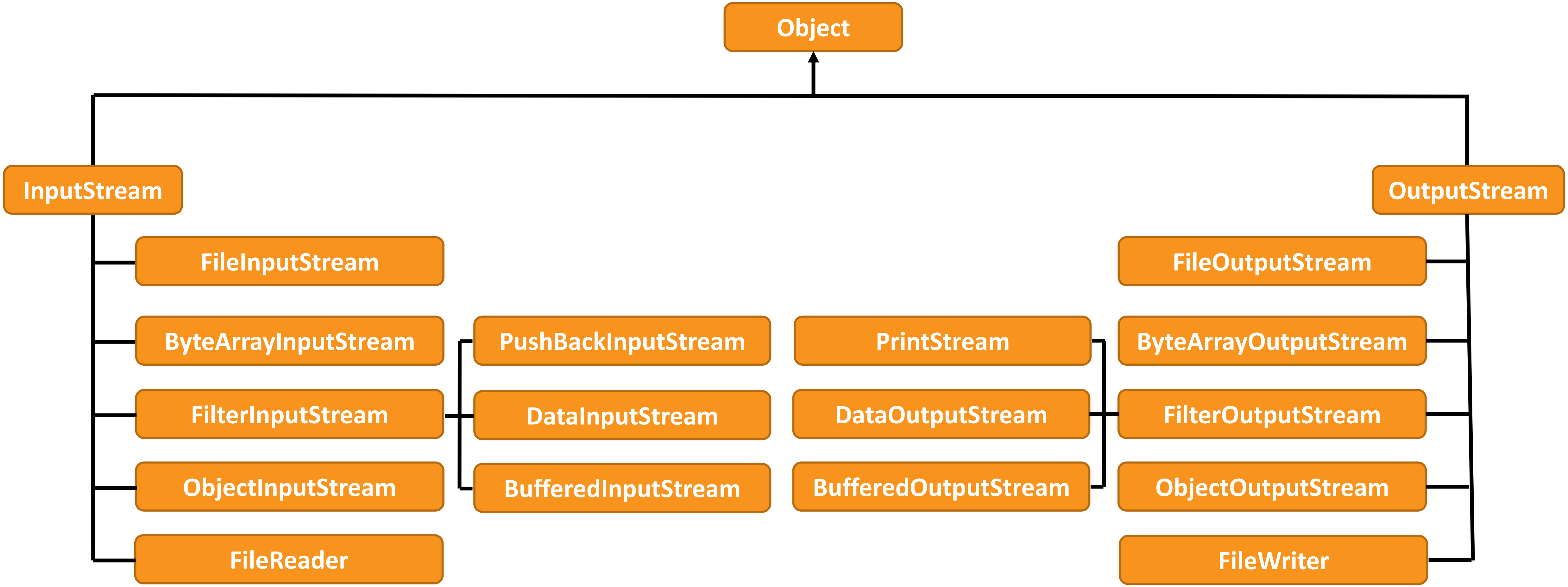
File Handling

architecture



File Handling

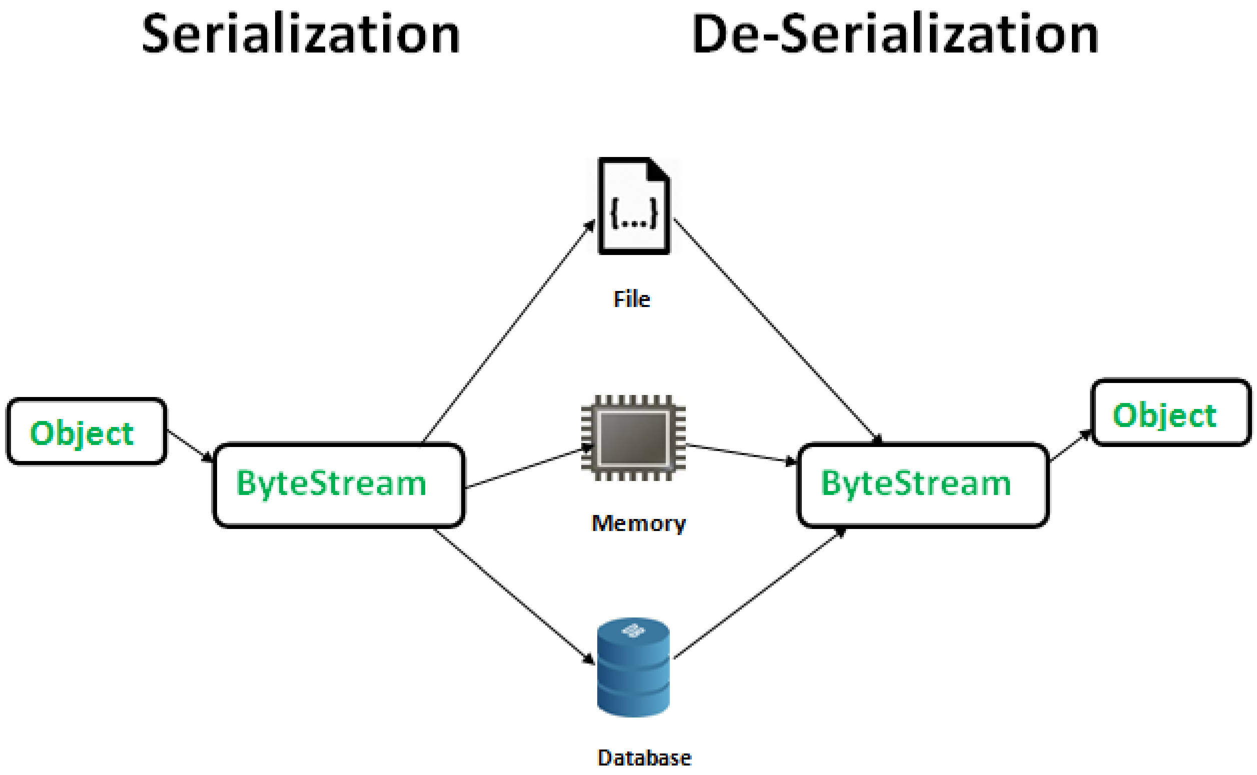
hierarchy

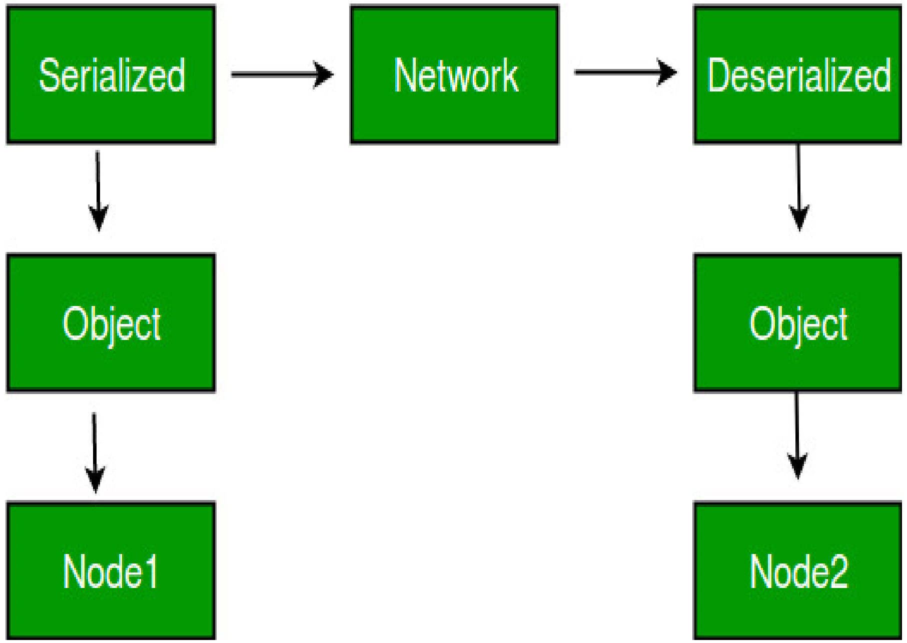


Serialization & Deserialization

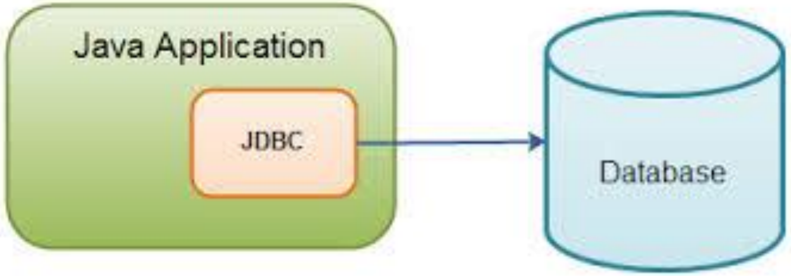
- **Serialization** is a mechanism of converting the state of an **object** into a **byte stream**.
- **Deserialization** is the reverse process where the **byte stream** is used to **recreate** the **actual Java object** in memory.
- This mechanism is used to **persist the object**.
- The byte stream created is **platform independent**. So, the object serialized on one platform can be deserialized on a different platform.
- To make a Java object serializable we **implement the java.io.Serializable** interface.
- The ObjectOutputStream class contains **writeObject()** method for serializing an Object.
- The ObjectInputStream class contains **readObject()** method for deserializing an object.

Serialization & Deserialization architecture

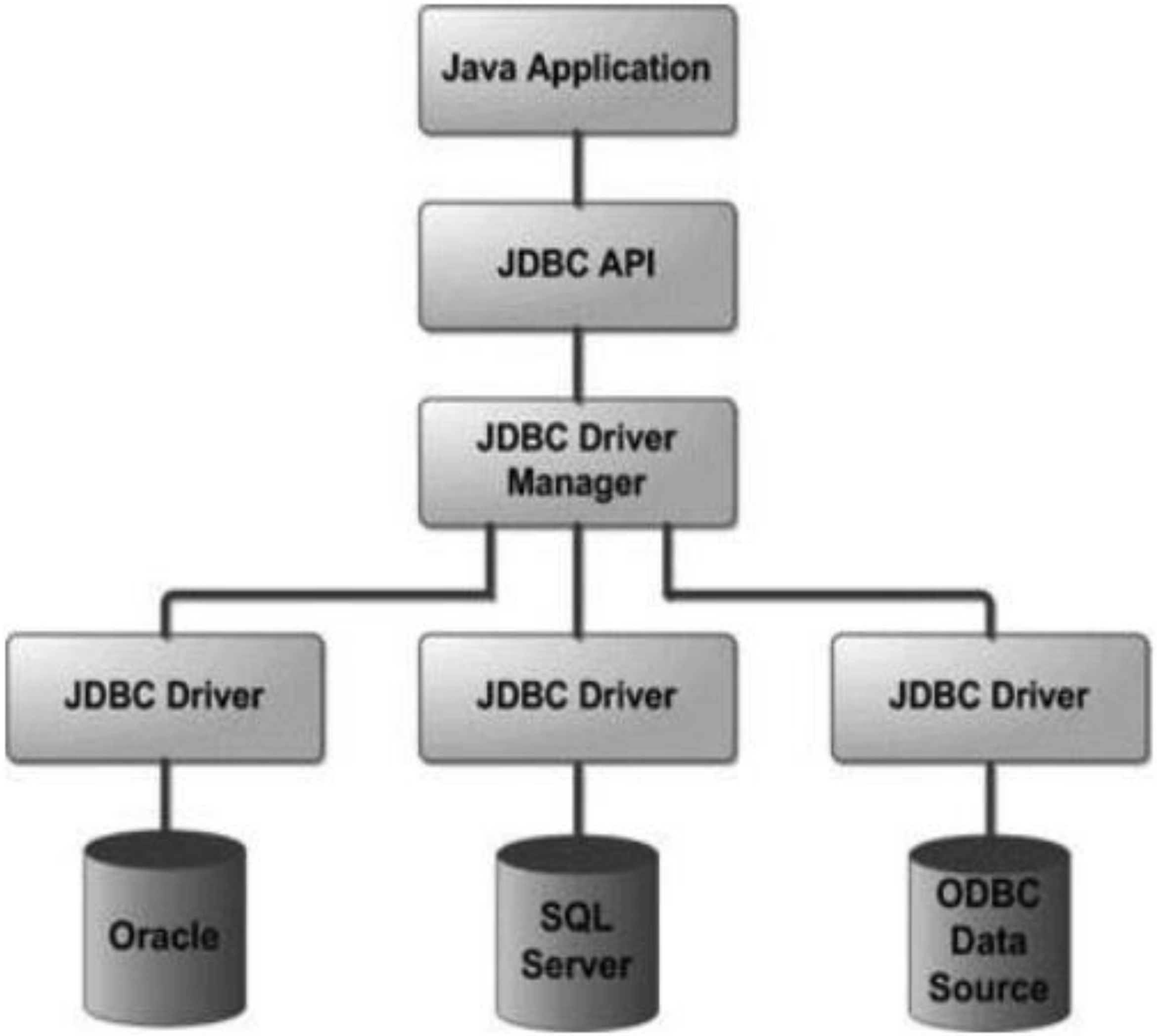




JDBC (Java Database Connectivity)



- JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
- Vendor Specific driver package implements all the interfaces of Java’s JDBC API.
- Methods in JDBC API are capable in generating SQLException.



JDBC Architecture

- The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers
- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.
- The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

JDBC (Java Database Connectivity)

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

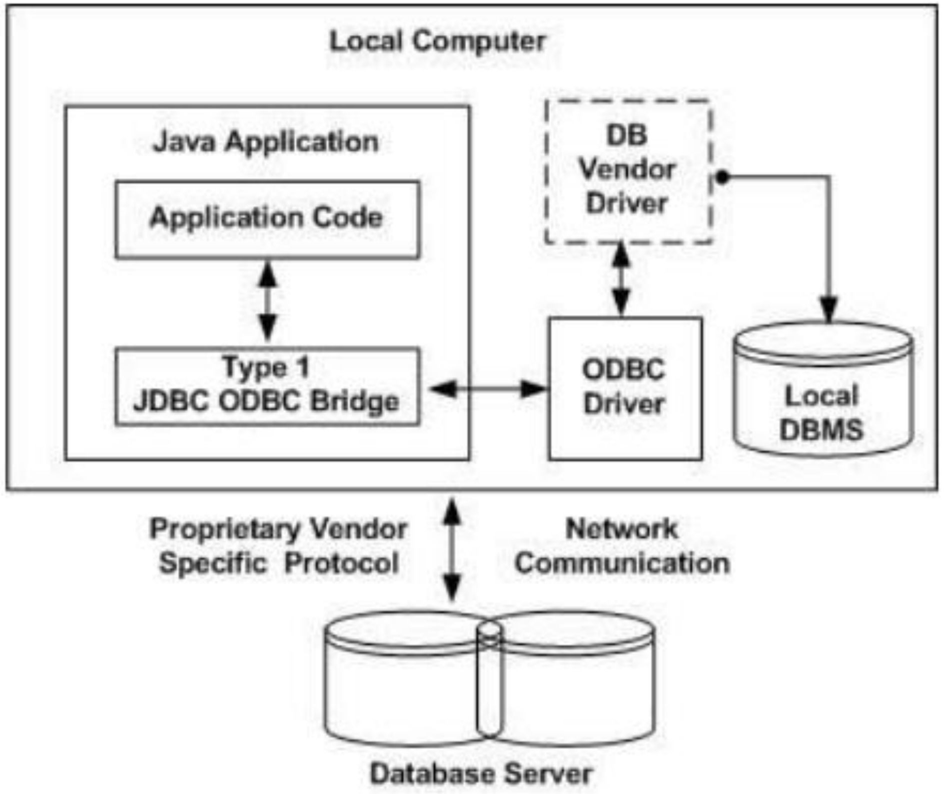
- **SQLException:** This class handles any errors that occur in a database application.

Type 1: JDBC-ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.

Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



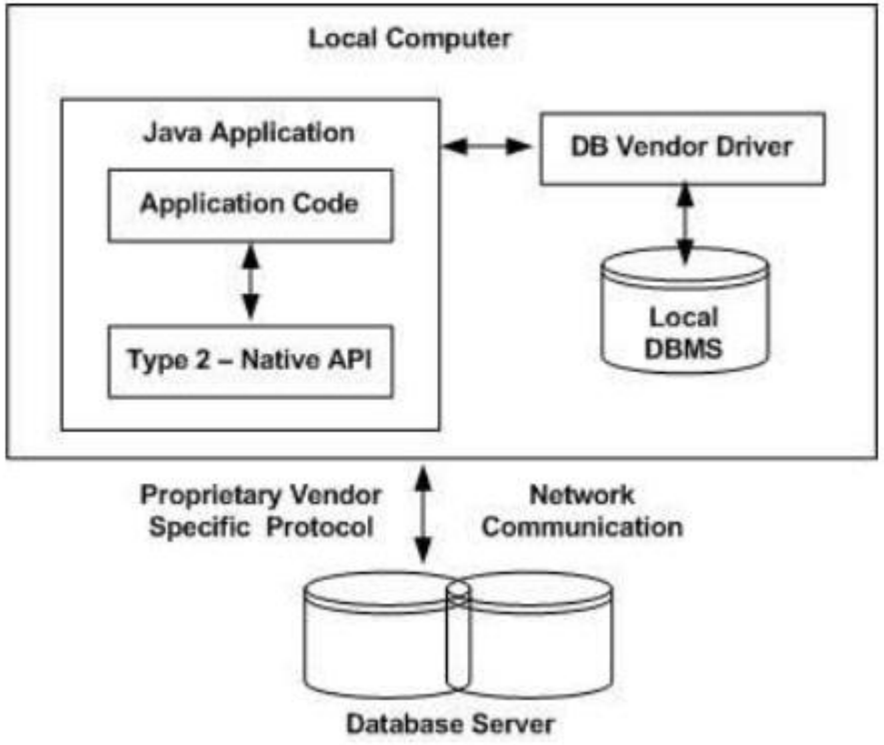
Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database.

These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.

The vendor-specific driver must be installed on each client machine.

If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.



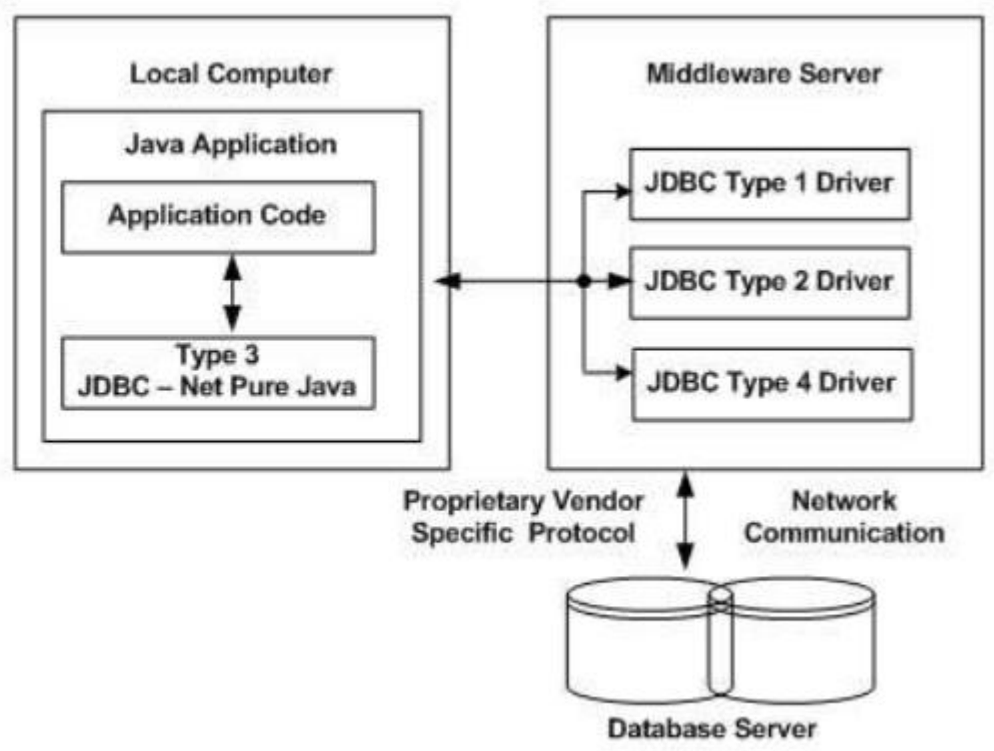
Type 3: JDBC-Net pure Java Driver

In a Type 3 driver, a three-tier approach is used to access databases.

The JDBC clients use standard network sockets to communicate with a middleware application server.

The socket information is then translated by the middleware application server into the call format required by the DBMS and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



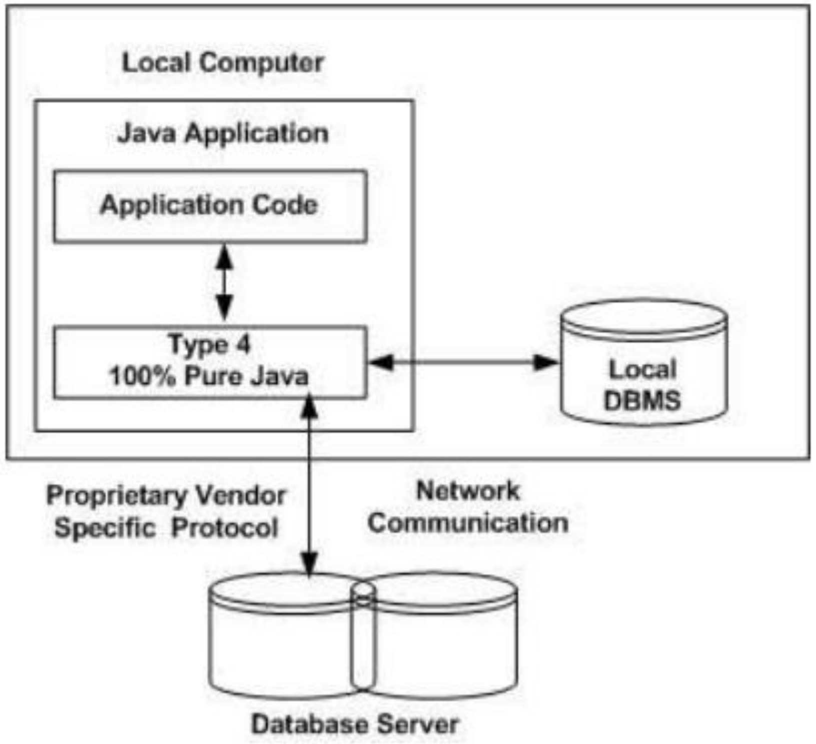
Type 4: 100% Pure Java Driver

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection.

This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server.

Further, these drivers can be downloaded dynamically.



7 Setps of JDBC

Step 1 : Import required packages.

Step 2 : Load and register Driver, (Create instance of Driver class).

Step 3 : Create connection with database using url, username & password.

Step 4 : Create Statement for executing query.

Step 5 : Fire the query on database, i.e. make an SQL request.

Step 6 : Retrieve result and process it.

Step 7 : Close connection with database.

Driver & URL format

Vendor Specific

Sr. No.	RDBMS	JDBC Driver Name	URL Format
1.	MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**host:port/databaseName
2.	Oracle	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**host:port:databaseName
3.	DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port/databaseName
4.	Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**host:port/databaseName

DriverManager interface

methods for creating Connection

```
public static Connection getConnection(String url)


public static Connection getConnection(String url, Properties prop)


public static Connection getConnection(String url, String un, String pwd)
```

ResultSet interface

constants

Sr. No.	public static final (constant)	Description
1.	TYPE_FORWARD_ONLY	The constant indicating the type for a ResultSet object whose cursor may move only forward.
2.	TYPE_SCROLL_INSENSITIVE	The constant indicating the type for a ResultSet object that is scrollable but generally not sensitive to changes to the data that underlies the ResultSet.
3.	TYPE_SCROLL_SENSITIVE	The constant indicating the type for a ResultSet object that is scrollable and generally sensitive to changes to the data that underlies the ResultSet.
4.	CONCUR_READ_ONLY	The constant indicating the concurrency mode for a ResultSet object that may NOT be updated.
5.	CONCUR_UPDATABLE	The constant indicating the concurrency mode for a ResultSet object that may be updated.
6.	CLOSE_CURSORS_AT_COMMIT	The constant indicating that open ResultSet objects with this holdability will be closed when the current transaction is commited.
7.	HOLD_CURSORS_OVER_COMMIT	The constant indicating that open ResultSet objects with this holdability will remain open when the current transaction is commited.

Connection interface

methods for creating Statement & PreparedStatement

```
public Statement createStatement()

public Statement createStatement(int move, int read)

public Statement createStatement(int move, int read, int cursor)


public PreparedStatement prepareStatement(String q)

public PreparedStatement prepareStatement(String q, int move, int read)

public PreparedStatement prepareStatement(String q, int move, int read,
                                          int cursor)
```

Statement interface

methods

1) **public ResultSet executeQuery(String sql) throws SQLException**

   This method is used to execute SELECT query. It returns the object of ResultSet.

   (DQL Queries)

2) **public int executeUpdate(String sql) throws SQLException**
   This method is used to execute specified query, it may be create, drop, insert, update, delete etc. **(DDL & DML Queries)**

3) **public boolean execute(String sql) throws SQLException**
   This method is used to execute queries that may return multiple results.

4) **public int[] executeBatch() throws SQLException**
   This method is used to execute batch of commands. **(DML Queries)**

ResultSet interface

methods for navigating ResultSet

```
public void beforeFirst() throws SQLException
public void afterLast() throws SQLException
public boolean first() throws SQLException
public boolean last() throws SQLException
public boolean previous() throws SQLException
public boolean next() throws SQLException
public boolean absolute(int row) throws SQLException
public boolean relative(int row) throws SQLException

public int getRow() throws SQLException
public boolean isBeforeFirst() throws SQLException
public boolean isAfterLast() throws SQLException
public boolean isFirst() throws SQLException
public boolean isLast() throws SQLException
```

ResultSet interface

methods for retrieving data from ResultSet

```
public int getInt(String columnName) throws SQLException
public int getInt(int columnIndex) throws SQLException

public String getString(String columnName) throws SQLException
public String getString(int columnIndex) throws SQLException

public double getDouble(String columnName) throws SQLException
public double getDouble(int columnIndex) throws SQLException
```

ResultSetMetaData interface

methods

1) **public int getColumnCount() throws SQLException**

   This method returns the total number of columns in the ResultSet object.

2) **public String getColumnName(int index) throws SQLException**
   This method returns the column name of the specified column index.

3) **public String getColumnTypeName(int index) throws SQLException**
   This method returns the column type name for the specified index.

4) **public String getTableName(int index) throws SQLException**
   This method returns the table name for the specified column index.

DatabaseMetaData interface

methods

1) public String getDriverName() throws SQLException:

This method returns the name of the JDBC driver.

2) public String getDriverVersion() throws SQLException:

This method returns the version number of the JDBC driver.

3) public String getUserName() throws SQLException:

This method returns the username of the database.

4) public String getDatabaseProductName() throws SQLException:

This method returns the product name of the database.

5) public String getDatabaseProductVersion() throws SQLException:

This method returns the product version of the database.

6) public ResultSet getTables(String cat, String sch, String tn, String[] types) throws SQLException:

It returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, TABLE, etc.

Thank You