Java Language Fundamentals

1. Identifiers
2. Reserved Words
3. Data Types
4. Literals
5. Arrays
6. Types of variables
7. Var-arg methods
8. Main method
9. Command-Line Arguments
10. Java-Coding Standards

***1. Identifiers:*** Which can used for identification. A name in java program is called as identifiers and can be used for identification. It can method name, class name, variable name, label name.

class Test { // class name identifier

public static void main(String[] args) { //method name, String, args

int x = 10; //name of the variable

}

}

*Rules for defining java identifiers:*

1. The only allowed character in java identifiers are a - z, A - Z, 0 - 9, $, \_. If we are using any other character, we will get compile-time error

Example: total\_number - valid

total# - invalid

2. Identifiers can’t starts with digits

Example: total123 - valid

123total - invalid

3. Java identifiers are case sensitive, of course java-language itself as case-sensitive programming language.

class Test {

int number = 10;

int Number = 10;

int NUMBER = 10; // this is valid

}

4. No length limit for java identifiers

5. We can’t use reserved words as identifiers

6. All predefined java class-name and interface names, we can use as identifiers. But this kind of of programs are not recommended.

*1. Reserved words:*

In java, some words are reserved, to represent some meaning or functionality, such type of words are called reserved words.

Reserved Word(53)

Reserved Literals(3) - true, false, null

Keywords(50)

unused (2) - goto, const

used(48) -

Keywords for data-type(8) - byte, short, int, long, float, double, boolean, char

Keywords for flow-control(11) - if, else, switch, case, default, while, do, for, break, continue, return

Keywords for modifiers(11) - private, public, protected, static final, abstract, synchronized, native, strictfp, transient, volatile

Keywords for exception-handling(6) - try, catch, finally, throw, throws, assert

Class related keyword(6) - class, interface, extends, implements, package, import

Object related keyword(4) - new, instaceof, super, this

Return-type keyword - void

Enum keyword - enum

1. All 53 reserved-words in java contains only lower-case alphabet symbols.
2. In java we have only new keyword and there is no delete keywords, because of garbage collector.
3. strictfp, assert, enum as added as part of from 1.5 v

*2. Data-Types*

In java, every variable and every expression has some type. Each and every datatype is clearly defined. Every assignment should be checked by compiler for type compatibility. Because of above reason, we can conclude, java language is strongly-typed programming language.

Java is not considered as pure OO programming language, because several OOPS features are not satisfied by java, like Operator Overloading, Multiple Inheritance, etc. Moreover, we are depending on primitive data type which are non-objects.

*Primitive datatypes(8):-*

*Numeric data-type*

Integral data-type: byte, short, int, long

Floating Point data-type: float, double

*Non-numeric data-type*: char, boolean

Except boolean and char, remaining data-types are considered as signed data-types, because we can represent both positive and negative numbers.

*byte:**Size: 1 byte(8 bits)*

*+/- 1 1 1 1 1 1 1*

*MAX\_VALUE : +127*

*MIN\_VALUE : -128 (Because of 2’s complement representation)*

*0000000*

*Range: -128 to 127*

Positive numbers will be represented directly in the memory, whereas negative number will be represented as 2’s complement form.

*Invalid*

byte b = 128; ===> will throw Possible loss of precision found : int

Required: byte

byte b = 10.5; ===> same error will be thrown

byte b = true; ===> Incompatible type found

Byte is the best choice, if you want to handle data in term of streams, either from the file or from the network(file supported or network support only byte)

*short:* short is most rarely used data-type in java.

Size: 2 byte(16 bits)

Range: -2 15 to 2 15 -1 [-32768 to 32767]

short s = 32768; ===> will throw PLP found : int

Required: short

short s = 10.5; ===> same error will be thrown

short s = true; ===> Incompatible type found

Short data-type best suitable for 16-bit processor like 8085, but these processor are completely out-dated and hence corresponding data-type is also outdated datatype.

*int*: The most common datatype

Size: 4 bytes(32 bits)

Range: - 2 31 to 2 31 -1

*long:* Sometimes int may not enough to hold bigger values, then we should go for long type.

Ex 1: distance travelled by light in thousand days.

Ex 2: to calculate the number of character present in a file

Size: 8 bytes(64 bits)

Range: -2 63 to 2 63 -1

*Floating point data type:-*

Float: 5 to 6 place of accuracy

Double: 14 to 15 place of accuracy

Float: Single precision

Double: Double precision

Float: Size 4 bytes -1.7 e38 to 1.7 e38

Double: 8 bytes - 3.4 e38 to -3.4 e38

*boolean:-* Size: Not applicable, it depend on VM

Range: true to false

*char:-* Size: 2 bytes

=================================================

*Literals:* A constant value which can assigned to a variable is called literal.

int x = 10; // 10 is a constant value or literal

*Integral Literal:-* For integral datatypes(byte, short, int, long) we can specify literal value in the following ways.

Decimal Literal(base 10) (0-9)

Octal form(base 8) (0-7)

Hexa form(base 16) (0-9, A-F, a-f)

int x = 10;

int x1 = 0777;

int x2 = 0789; //invalid

int x3 = 0Xface;

int x4 = 0xFACE;

int x5 = 0xBEEF;

int x6 = 0XBEER; //invalid

Programmer is having the choice to specify the number in decimal, octal, hex but JVM will print that in decimal.

int x7 = 10;

int x8 = 010;

int x9 = 0X10;

System.out.println(x7+" "+x8+" "+x9); //10 8 16

By default, every integral literal is of integer type, but we can explicitly specify it in long by suffixed with small ‘l’ or ‘L’

int i = 10;

int i1 = 10l; //invalid 8 bytes to 4 bytes

long l = 10;

long l = 10l;

There is no direct way to specify byte and short literals explicitly, but indirectly we can specify. Whenever we are assigning integral literal to the byte variable and the value within the range of byte, then the compiler treat it automatically as byte literal. Similiarly short literal also.

Eg: byte b = 10;

byte b = 127;

byte b = 128 // PLP

short s = 32767;

short s = 32768; //PLP (Possible loss of precision)

*Floating point Literal:-*

By default every floating literal is of double type and hence we can’t assign directly to the float variable. But we can specify floating point literal as float type by suffixed with small ‘f’ or ‘F’

float f = 123.456 //PLP, ‘cause it is double type

float f = 123.456f;

double d = 123.456;

double d = 123.456D;

double d1 = 123.456d;

We can specify floating point literal as double type by suffixed with small ‘d or ‘D’ of course this convention is not required.

double d1 = 123.456d;

float f1 = 123.456d; //invalid PLP

We can specify floating point literal only in decimal, and we can’t specify in octal and hexa- decimal forms.

double d2 = 123.456;

double d3 = 0123.456; //This is not octal, it is decimal

double d4 = 0X123.456; //invalid

System.out.println(d2+" "+d3);

We can assign integral literal directly to floating point variables and that integral literal can be specified either in decimal or octal or hexadecimal forms.

double d4 = 0786; //will be considered as octal

double d5 = 0786.0; // will be considered integral as floating point is there

double d6 = 0XFace; //will be considered as integral

double d7 = 0XFace.0; //no not possible will as not considered as integral

We can’t assign floating point literal to integral types.

double d = 10;

int x = 10.0; //invalid

We can specify floating point literal, even in exponential form(scientific notation)

double d8 = 1.2e3;

float f1 = 1.2e3; //invalid

float f2 = 1.2e3f;

System.out.println(f1);

*Boolean literal:*

Only allowed values: true/false

boolean b = true;

boolean b1 = 0; //invalid incompatible types

boolean b2 = True; //invalid, can’t find variable

boolean b3 = "True"; //invalid incompatible types

*Char literal:*

We can specify char literal as single character in single quotes.

char ch = ‘a’;

char ch = a; //invalid, cannot find symbol a

char ch = “a”; //invalid Incompatible type found string, required char

char ch = ‘ab’;//invalid unclosed char literal

We can specify char literal as integral literal which represents unicode value of the character and that integral literal can specified in either in decimal, octal or in hexa decimal forms. But allowed range is 0 to 65535

char ch = 97;

char ch = 0XFace;

char ch = 0777;

char ch = 65535;

char ch = 65536; //invalid

We can represent char literal in unicode code representation, which is nothing but ‘\uXXXX’

char ch2 = '\u0061';

System.out.println(ch2); //a

Every escape character is valid char literal.

char ch3 = '\n';

char ch4 = '\t';

char ch5 = '\m'; //illegal escape character

Escape Characters

\n -> newline

\t -> horizontal tab

\r -> carriage return

\b -> back space

\f -> form feed

\’ -> single quote

\” -> double quote

\\ -> Back slash

*String literal:* Any sequence of character within double quotes is treated as String literal.

Eg: String s = “Suganthan”;

In 1.7 enhancement with respect to literals

*Binary Literal*

For integral datatype until 1.6v we can specify literal value in the following ways

Decimal

Octal

Hexa

But from 1.7v onwards we can specify literal values even in binary form also.

Allow digits are 0 & 1. Literal value should be prefixed with ‘0b’ or ‘0B’

*Usage of underscore literal;*

From 1.7 onwards usage of underscore in between is allowed.

double d = 123456.789;

double d1 = 1\_23\_45.7\_8\_9;

Readability will improved.

Compiler will delete all these underscores during compilation

Any number of underscores are allowed

*Invalids:*

double d = \_1\_23\_45.7\_8\_9

double d = 1\_23\_45\_.7\_8\_9;

double d = 1\_23\_45.7\_8\_9\_

byte 1b -> short 2b -> int 4b -> long 8b -> float 4b -> double 8b

^

|

char -> |

8b long value we can assign to 4b float variable because, both are following different memory representations internally.

float f = 10l;

sop(f); //10.0

*Arrays:*

1. Introduction
2. Array declaration
3. Array creation
4. Array Initialization
5. Array declaration, creation & initialization in a single line
6. length vs length()
7. Anonymous arrays
8. Array element assignments
9. Array variable assignments

*Array Introduction:*

An array is an indexed collection of fixed number of homogenous data elements.

The main advantages of arrays is, we can represent huge number of values by using single variable. So that readability of the code will be improved.

But the main disadvantages of arrays is fixed in size. Once we creates an array, there is no chance of increasing or decreasing the size based on our requirement. Hence to use arrays concept, compulsory we should know the size in advance, which may not possible always.

*Array Declaration:*

*One Dimensional Array Declaration:-*

*Ways to declare*

int[] x; //This recommended, because name is clearly separated from type

int []x;

int x[];

At the time of declaration, we can’t specify the size, otherwise we will get compile time error.

int[6] x; //invalid

Size is required during creation.

*Two dimensional array declaration:-*

*Valid declarations:-*

int[][] x;

int [][]x;

int x[][];

int[] []x;

int[] x[];

int x[][];

Which of the following are valid

int[] a,b; //a -> 1, b ->1

int[] a1[],b1; //a -> 2, b ->1

int[] a2[],b2[]; //a -> 2, b ->2

int[] []a3,b3; //a -> 2, b ->2

int[] []a4,b4[]; //a -> 2, b ->3

int[] []a5,[]b5; //invalid

If you want to specify dimension before the variable that facility is applicable only for first variable in declaration. If we are trying to apply for next variables onwards, we will get compile time error.

For example:

int[] []a, []b, []c;// here b and c are invalid declaration

*Three dimensional array declaration:*

*Valid declaration styles:*

int[][][] a6;

int [][][]a7;

int a8[][][];

int[] [][]a9;

int[] []a10[];

int[] a11[][];

int[][] []a12;

int[][] a13[];

int [][]a14[];

int []a15[][];

*Array creation:*

Every array in java is an object only, hence we can create arrays by using ’new’ operator.

int[] a = new int[3];

Every array type corresponding classes are available, and these class are part of java language and not available to the programmer level.

int []a15[][] = new int[1][1][1];

System.out.println(a15.getClass().getName()); //[[[I

|  |  |
| --- | --- |
| Array type | Corresponding class name |
| int[] | [I |
| int[][] | [[I |
| double[] | [D |
| short[] | [S |
| byte[] | [B |
| boolean | [Z |

At the time of array creation, compulsory we should specify the size.

int[] x = new int[]; //invalid

It is legal to have an array with size 0 in java.

int[] x10 = new int[0];//perfectly ok

If we trying to specify array size with some negative int value, then we will Runtime exception saying, negativeArraySizeException.

int[] x = new int[-3];//RE: NegativeArraySizeException

To specify array size, the allowed data-types are byte, char, short, int. If we are trying to specify any other type then, we will get compile time error.

int[] x12 = new int[1];

int[] x13 = new int['a'];

byte b5 = 10;

int[] x14 = new int[b5];

short s = 30;

int[] x15 = new int[s];

//int[] x16 = new int[10l]; invalid

The maximum allowed array size in java is 2147483647(int max), which the maximum value of int datatype.

int[] x16 = new int[2147483647];

int[] x17 = new int[2147483648]; //out of range

Even in the first case, we may get runtime exception, if sufficient heap memory not-available.

*Two Dimensional Array creation:-*

In java 2-dimensional array is not implemented by using matrix. Sun people followed **array of arrays** approach for multi-dimensional array creation. The main advantage of this approach is, memory utilization will be improved.

Example 1: int[][] x = new int[2][];

x[0] = new int[2];

x[1] = new int[3];

Example 2: int[][][] x = new[2][][];

x[0] = new int[3][];

x[0][0]=new int[1];

x[0][1] = new int[2];

x[0][2] = new int[3];

x[1] = new int[2][2];

Without specifying the base value, we can’t able to specify the next level size of an array.

*Array initialization:*

Once we creates an array, every element by default initialized with default values.

int[] x17 = new int[1];

System.out.println(x); //[I@15db9742

System.out.println(x[0]); //0

Note: whenever we are trying to print any reference variable, internally toString() method will be called, which is implemented by default, to return the string in the following form.

classname@hashCodeinhexadecimal

*Array declaration, creation and initialization in a single line:*

We can declare, create and initialize an array in a single line

int[] x = {10, 20, 30};

char[] ch = {‘a’, ‘e’, ‘i’, ‘o’, ‘u’};

String[] str = {“a”, “aa”, “aaa”};

*For two dimensional:*

We can extend this shortcut for multi-dimensional arrays also.

int[][] x = {{10, 20}, {30,40,50}}

For three dimensional

int[][][] x18 = {

{ { 10,20,30}, {40,50,60} },

{ {20,80 }, {90,80,110} }

};

System.out.println(x18[0][1][2]); //60

System.out.println(x18[1][0][1]); //80

System.out.println(x18[1][1][1]); //80

If we want to use this shortcut, compulsory we should perform all activities in a single line. If we are trying to divide into multiple lines, then we will get compile time error.

int[] x19;

//x19 = {10, 20, 30}; invalid

*length vs length():*

*Length:*

Length is a final variable, applicable for arrays. Length variable represents the size of the array.

int[] x = new int[6];

sop(x.length()) //CE error

sop(x.length) //6

Length method is a final method applicable for String objects and length method returns number of characters present in the String.

String str = "Suganthan";

str.length();

In multidimensional array, length variable represents only base size, but not total size.

int[][] x20 = new int[6][3];

System.out.println(x20.length); //6

System.out.println(x20[0].length); //3

System.out.println(x20[1].length); //3

There is no direct way to find the length of multidimensional arrays. But indirectly we can find as follow.

x20[0].length + x20[1].length + …

*Anonymous Arrays:*

Sometimes we can declare an array without name, such type of nameless arrays are called anonymous arrays.

The main purpose of anonymous arrays is just for instant use(one time usage).

display(new int[]{10, 30, 40});

*Array element assignments:*

In the case of primitive type array, we can provide any type which can be implicity promoted to declared type

int[] x12 = new int[1];

int[] x13 = new int['a'];

byte b5 = 10;

int[] x14 = new int[b5];

short s = 30;

int[] x15 = new int[s];

In case of float type arrays, the allowed data-types are, byte, short, char, int, float.

In case of object type arrays, itself and its child classes are allowed.

Number ->

Its child byte, short, int, long, float, double.

*Array variable assignment*

*Case 1:* Element level promotion is applicable to array type.

int[] x = {10, 20, 30};

char[] ch = {‘a’, ‘b’, ’c’};

int[] b = x; //valid

int[] c = ch; //invalid

Char element can be promoted to int element, but I can’t promote char array to int array.

Char -> int //valid

Char[] -> int[] //invalid

Int -> double //valid

Int[] -> double[] //invalid

Float -> int //invalid

Float[] -> int[] //invalid

String -> Object //invalid

String[] -> Object[] //valid

If it is object type arrays, child class type array can be promoted to Object arrays.

String[] str1 = {"a", "b", "c"};

Object[] obj = str1;

*Case 2:*  Whenever we are assigning one array to another array, internal elements won’t be copied, just reference variables will the reassigned.

int[] x21 = {10, 20, 30, 40, 50, 60};

int[] x22 = {70, 80};

x21 = x22; //valid

x22 = x21; //valid

*Case 3:* Whenever we are assign one array to another array, the dimensional must be matched. For example in the place of one-dimensional int array, one dimensional array only. If we are trying to provide any other dimesion, then we will get compile-time error.

int[][] x23 = new int[3][];

//x23[0] = new int[4][3]; invalid

//x23[0] = 0; invalid

x23[0] = new int[10];

Whenever one array to another array, both dimensions and types must be match, but sizes are not required to match.

----------------------------------------------------------------------------------------------

class Test {

public static void main(String[] args ){

for(int i=0; i<args.lenth;i++) {

sop(args[i]); //will lead to ArrayIndexOutOfBoud

}

}

}

------------------------------------------------------------------------------------------------

class Test {

public static void main(String[] args) {

String[] argh = {“x”, “y”, “z”};

args = argh;

for(String s : args) {

sop(s); //will print x y z

}

}

}

---------------------------------------------------------------------------------------------

int[][] a = new int[4][3]; -> 5 object will get created

a[0] = new int[4]; -> 1

a[1] = new int[2]; -> 1

a = new int[3][2]; -> 4

Total : 11 objects got created

GC eligible 7 objects

*Types of variables:*

*Division 1:*

Based on type of value represented by a variable, all variables are divided into two types.

*Primitive variables:* Can be used to represent values

Ex: int x =10;

*Reference variables:* Can be used to refer objects

Ex: Student s = new Student(); //s is pointing to object

*Division 2:* Based on position of declaration and behavior all variables are divided into three category

1. Instance variables
2. Static variables
3. Local variables

*Instance variables:*

If the value of a variable is varied from object to object, such type of variables are instance variables.

For every object a separate copy of instance variables will be created.

Instance should be declared within the class directly, but outside of any method or block or constructor

Instance variable will be created at the time of object creation and destroyed at the time of object destruction, hence the scope of instance variable is exactly same as scope of object.

If object is there in memory scope of instance variable is there. If the object is destroyed, scope of instance variable gone.

Instance variable will be stored in the heap memory as the part of object.

We can’t access directly from static area. But we can access, by using object reference.

But we can access instance variable directly from instance area.

public class VariableTypeTest {

int x = 0;

public static void main(String[] args) {

//System.out.println(x); not possible

VariableTypeTest test = new VariableTypeTest();

System.out.println(test.x);

}

public void instanceMethod() {

System.out.println(x);

}

}

For instance variables, jvm will always provide default values and we are not required to perform initialization explicitly.

public class VariableTypeTest {

int x;

double d;

float f;

boolean b;

String str;

char c;

public static void main(String[] args) {

//System.out.println(x); not possible

VariableTypeTest test = new VariableTypeTest();

System.out.println(test.x); //0

System.out.println(test.d); //0.0

System.out.println(test.f); //0.0

System.out.println(test.b); //false

System.out.println(test.str); //null

System.out.println(test.c); //

}

}

Instance variables also known as object level variable or attributes.

Stored in heap memory.

*Static variables:*

If the value of a variable is not varied from object to object, then it is not recommended to declare variable as instance variable. We have to declare such type of variable by using static modifier.

In the case of instance variables, for every object a separate copy will be created, but in the case of static variables a single copy will be created at class level. And shared by every object of the class.

Static variables should be declared within the class directly but outside of any method or block or constructor.

Static variables will be created at the time of class loading and destroyed at the time of class unloading, hence scope of static variable is exactly same as scope of .class file.

*java Test*

1. Start JVM
2. Create and start main Thread
3. Locate Test.class file
4. Load Test.class -> static variable creation
5. Execute main() method
6. Unload Test.class -> static variable destruction
7. Terminate main Thread
8. Shutdown JVM

Static variable will be stored in method area.

Local variable will be stored in stack memory.

For static variables JVM will provide default values and we are not required to perform initializing explicitly.

Static variables are also known as class level fields.

Class Test {

Static int x = 10;

Int y = 20;

P s v main(String[] s) {

Test test = new Test();

Test.x = 888;

Test.y = 999;

Test test1 = new Test();

sop(test1.x+””+test1.y); //888 20

}

}

*Local Variables:-*

Sometimes to meet temporary requirements of the programmer, we can declare variables inside a method, inside a block or inside a constructor are called local variable.

It is also known are as temp/stack/automatic variable.

In memory, these variables are stored in stack memory.

The local variable will be created, while executing the block in which the local variable is declared and destroyed after completing that block.

For every execution, local variable will be created and destroy. Otherwise for every thread execution a separate copy will be created and destroyed. That is the season why it is thread safe. Hence the scope of local variable is the block in which we declared it.

For local variable, we should do the initialization by manually

public class VariableType1 {

public static void main(String[] args) {

// TODO Auto-generated method stub

try {

int j = Integer.parseInt("10"); //run-time error

}

catch(NumberFormatException e) {

j = 0; //compile-time error

}

System.out.println(j); //compile-time error

}

}

For local variables, jvm won’t provide default values compulsory we should perform initialization explicitly, before using that variable. I.e. If we are not using then, it is not required to perform initialization.

public class VariableType1 {

public static void main(String[] args) {

int x;

System.out.println("Hello"); //no issues

}

}

public class VariableType2 {

public static void main(String[] args) {

int x;

System.out.println("Hello"+x); //compile-time error.

}

}

It is not recommended to perform initialization for local variables inside any logical block. Because there is not guaranteed for the execution of these block always at runtime.

It is highly recommended to perform initialization at the time declaration atleast with default values.

The only applicable for local variables is final. By mistake if we are trying to apply any other modifier, then we will get compile time error.

public static void main(String[] args) {

private int x = 10; //CE

public int xx = 10; //CE

protected int xxx = 10; //CE

static int xxxx = 10; //CE

transient int xxxxx = 10; //CE

volatile int xxxxxx = 10; //CE

final int xxxxxxx = 10; //valid

}

If we are not declaring with any modifier, then by default it is default variable. But this rule is applicable only for instance and static variable but not for local variables.

Conclusions:

JVM will provide default initialization to instance and static variable but not for local variables

Instance and static variables can be accessed by multiple threads and hence these are not thread safe. But in the case of local variables for every thread, a separate copy will be created and hence local variables are thread-safe.

Every variable in the java should be either instance, static or local. Every variable in java should either primitive or reference.

Hence various possible combinations

Instance → primitive, Reference

Static → primitive, Reference

Local → primitive, Reference

*Uninitialized Arrays:*

We will get NPE(RE) for both instance and static variable.

int[] x; sop(x) is null, sop(x[0]) is null

int[] x = new int[3]; sop(x) is [I@12313, sop(x[0]) 0

In local level:

int[] x; sop(x) is CE, sop(x[0]) is CE

int[] x = new int[3]; sop(x) is [I@12313, sop(x[0]) 0

---------------------------------------------------------------------------------------------------------------

*Main method:*

Whether the class contains the main method or not, and whether the main method is declared according to requirement or not these things won’t be checked by compiler. At runtime JVM is responsible to check these things. If JVM unable to find main method then we will get runtime exception saying

“No such method error: main”

At runtime JVM always searches for the main method with the following prototype

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

The above syntax is very strict and if we perform any change and we will RE

Why it is public: to call JVM from anywhere

Why it is static: to call main without any existing object

Why it is void: main return won’t return anything to JVM

Why its name is main: Because it configured in JVM

String[] -> Command line argument.

Change which are allowed in main method

1. Can swap the access modifiers
2. String[] can replace with varargs String...

We can declare main with following modifiers

public class MainMethodTest {

***final synchronized strictfp*** public static void main(String... args) {

System.out.println("valid main method");

}

}

*Case 1:*

Overloading of the main method is possible. But JVM will always call String[] argument main method only.

public class MainMethodTest {

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

public static void main(int[] args) {}

}

Inheritance concept applicable for main method, hence while child class, if child doesn’t contain main method parent class main method will be executed.

public class MainMethodTest {

public static void main(String[] args) {

System.out.println("Parent");

}

}

public class ChildMethodTest extends MainMethodTest{}

*Method Hiding:*

public class MainMethodTest {

public static void main(String[] args) {

System.out.println("Parent");

}

}

public class ChildMethodTest extends MainMethodTest{

public static void main(String[] args) {

System.out.println("Child");

}

}

It seems method overriding concept applicable for main method, but it method hiding.