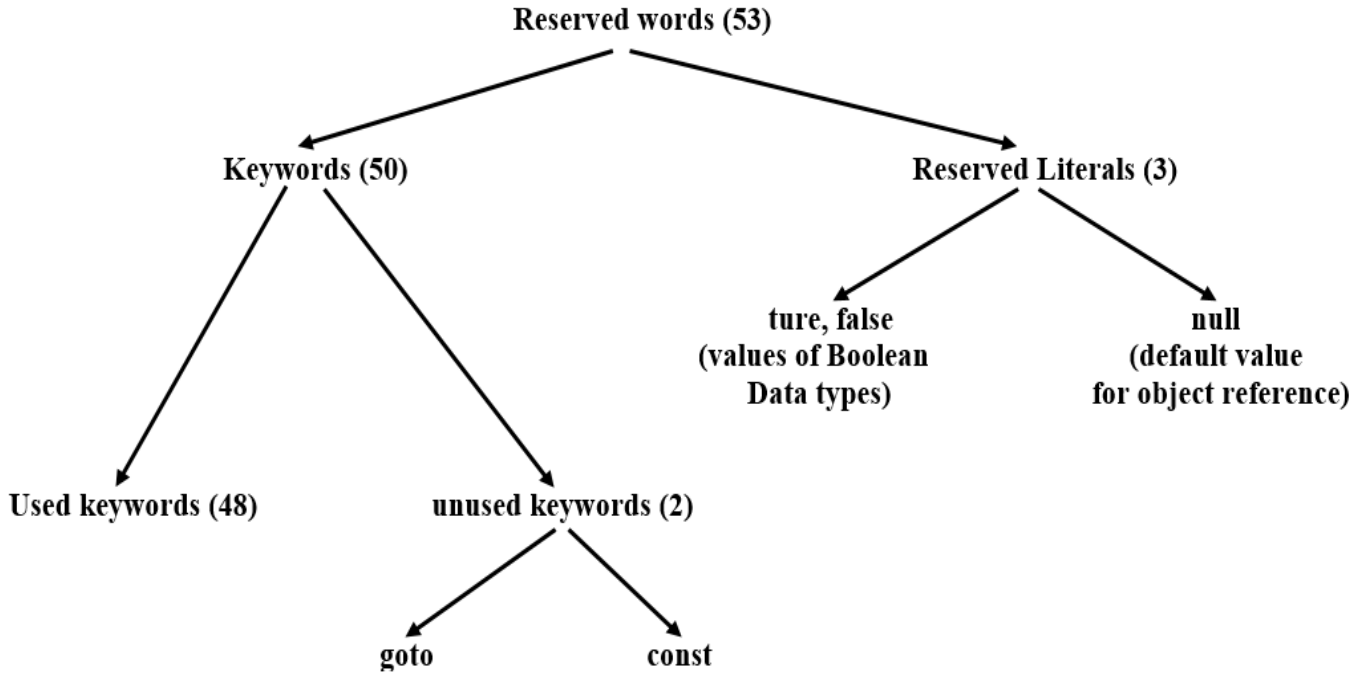
**Java language fundamentals**

**Identifier**
Any name given to java class, method, Variable and label in java program is called identifier.

**Example:**
```
class Test
{
        public static void main (String [] args)
        {
                int x=5;
        }
}
```

**Rules to define java identifiers:**
1. Allowed characters in java identifiers are:
    - a to z
    - A to Z
    - 0 to 9
    - -
    - $
2. Java identifiers are case sensitive. Even, java itself is case sensitive language.
3. There is no character length limit for identifiers.
4. Reserved words can't be used as identifiers.
5. All predefined java class names and interface names can be uses as identifiers.

**Reserved words**
Reserved words are associated to some predefined functionality or meaning such type of reserved identifiers are called reserved words.




**Reserved words for data types (8)**
Byte, short, int, long, float, double, char, boolean.

**Reserved words for flow control (11)**

If, else, switch, case, default, for, do, while, break, continue, return.

**Keywords for modifiers (11)**
Public, private, protected, static, final, abstract, synchronized, native, strictfp (l.2 version), transient, volatile.

**Keywords for exception handling (6)**
Try, catch, finally, throw, throws, assert (l.4 version)
**Class related keywords (6)**
Class, package, import, extends, implements, interface

**Object related keywords (4)**
New, instanceof, super, this.

**Void return type keyword**
If a method won't return anything compulsory that method should be declared with the void return type in java but is optional in C++.

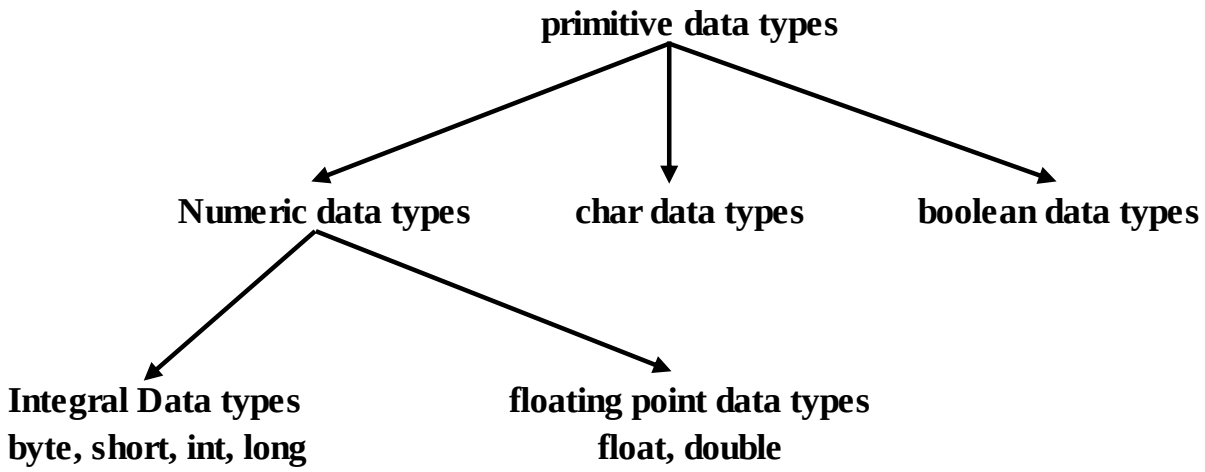
**Unused keywords**
**goto:** Not in use in java.
**const:** final is used instead of const.

**Reserved literals**
1) true
2) false
   (values for boolean data type).
3) Null (default value for object reference).

**enum**
This keyword was introduced in 1.5v to define a group of named constants

**Data types**
Every variable in java has a valid type, every expression has a type and all types are strictly define more over every assignment is checked by the compiler for type compatibility hence java is considered as strongly typed language.

Java is not considered as pure object-oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java and there is also dependency on primitive data types which are non-objects.




Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and"-ve" numbers.

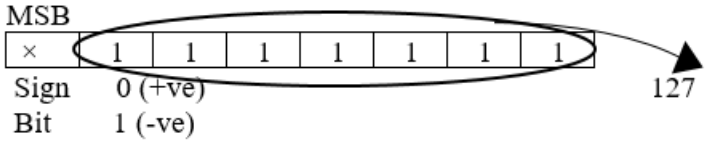
**Integral data types**

Integral data types are used to represent integer values. Data types (byte, short, int and long) can be used to represent whole numbers.

**Byte**

Size: 8bits

Range: -128to 127[$-2^7$ to $2^7$ -1]




1. The most significant bit acts as sign bit where "0" means "+ve" number and "1" means "-ve" number.
2. Positive numbers are represented directly in the memory whereas "-ve" numbers will be represented in 2's complement form.
3. byte data type is best suitable for handling data in terms of streams either from the file or network.

**Example**

```
byte b=l0;
byte b=130;          //C.E:possible loss of precision
byte b=l.5;          //C.E:possible loss of precision
byte b=true;         //C.E:incompatible types
```

**Short**

Short is rarely used data type in java.

Size: 2 bytes

Range: -32768 to 32767($-2^{15}$ to $2^{15}$-1)

**Example:**

```
short i=l0;
short i=32768;       //C.E:possible loss of precision
short i=true;        //C.E:incompatible types
```

**int**
int is Commonly used data type in java.
Size: 4 bytes
Range: -2147483648 to 2147483647 ($-2^{31}$ to $2^{31}-1$)

**Example**
int i=30;
int i=l0.5; //C.E: possible loss of precision
int i=true; //C.E: incompatible types

**long**
long is use when int is not enough to handle big value.
Size: 8 bytes
Range: $-2^{63}$ to $2^{63}-1$

**Floating Point Data types**
To represent real numbers floating point data types are used.
**float**
float data types is used represent real number with accuracy up to 5 or 6 decimal place.
Size: 4 bytes
Range: -3.4e38 to 3.4e38.

**double**
double data types is used represent real number with accuracy up to 14 or 15 decimal place.
Size: 8 bytes
Range: -l.7e308 tol.7e308.

**boolean**
Size and range Boolean vary from JVM to JVM but allowed values are either true or false.

**Example**
boolean b=true;
boolean b=True; //C.E:cannot find symbol
boolean b="True"; //C.E:incompatible types
boolean b=0; //C.E:incompatible types

**Char**
java is Unicode based to represent any worldwide alphabets/character.
Size: 2 bytes
Range: O to 65535

**Example**
char ch1=97;
char ch2=65536; //C.E:possible loss of precision

**Literals**
Any constant value assigned to the variable is called literal.
**Example:**
int x=10

**Integral Literals**

For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

**Decimal literals**

- Allowed digits are 0 to 9.
  Example: int x=10;

**Octal literals**

- Allowed digits are 0 to 7. Literal value should be prefixed with zero.
  Example: int x=010;

**Hexa Decimal literals**

- The allowed digits are 0 to 9, A to Z.
- For the extra digits we can use both upper case and lower-case characters.
- Hexadecimal literal value should be prefixed with ox(or)oX.
  Example: int x=0x10;

By default every integral literal is int type but we can specify explicitly as long type by suffixing ‘l’ or ‘L’.

**Example:**

```
Int x=10;
long l=10L;
long l=10;
int x=10l;                  //C.E:possible loss of precision(invalid)
```

If integral literal assigned to Byte variables are within the range of byte compiler automatically treats them byte literal. Similar concept is for short literal also.

**Example:**

```
byte b=10;                  (valid)
byte b=130;                 //C.E:possible loss of precision(invalid)
short i=32767;              (valid)
short i=32768;              //C.E:possible loss of precision(invalid)
```

**Floating Point Literals**

Floating point literals are by default double but can be specified explicitly as float by suffixing ‘f’ or ‘F’.

**Example:**

```
float f=123.456;            //C.E:possible loss of precision(invalid)
float f=123.456f ;          (valid)
double d=123.456;           (valid)
```

We can specify explicitly floating-point literal as double type by suffixing ‘d’ or ‘D’.

**Example:**

```
double d=123.456D;
```

We can specify floating point literal only in decimal (octal and hexadecimal not allowed).

**Example:**

```
double d=123.456;           (valid)
double d=0123.456;          (valid)
double d=0x123.456;         //C.E:malformed floating point literal(invalid)
```

**Boolean literals**

The only allowed values are either ‘true’ or ‘false’.

Example:

```
boolean b=true;          (valid)
boolean b=0;             //C.E:incompatible types(invalid)
boolean b=True;          //C.E:cannot find symbol(invalid)
boolean b="true";        //C.E:incompatible types(invalid)
```

**Char literals**

A char literal are represented as single character within single quotes.

**Example:**

```
char ch='a';             (valid)
char ch=a;               //C.E:cannot find symbol(invalid)
char ch="a";             //C.E:incompatible types(invalid)
char ch='ab';            //C.E:unclosed character literal(invalid)
```

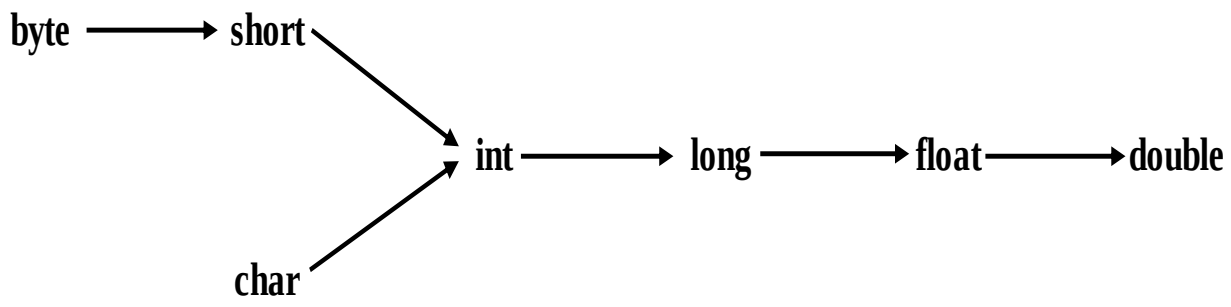
char literal can be integral literal (either in decimal or octal or hexadecimal) which will represent Unicode of that character. But allowed values must be is range 0 to 65535.

**String Literals**

sequence of characters in double quotes is treated as String literal.

**Example**

```
String s="CSE";          (valid)
```




**Variable**

In java Variables are divided into two types based upon the value they represent and they are:

**Primitive variables**

Primitive variables are used to represent primitive values.

**Example**

```
int x=5;
```

**Reference variables**

Reference variables are used to refer objects.

**Example**

```
Student s=new Student ();
```

Based upon the position of declaration and purpose all variables are divided into the three types.

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

**Instance variable**

- A variable whose value is varied from object to object is called instance variables
- A separate copy of instance variables is created for each object.

- Instance variables is created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is same as scope of objects.
- Instance variables are stored in the heap area.
- Instance variables are declared inside the class, but outside of any method, block or constructor.
- We can access Instance directly from static area. But by using object reference we can access instance variables from static area.
- Instance variables also known as object level variables or attributes.

**Static variables**

- If it is not required for variable to vary its value from object to object then variables are not recommended to declare as instance variables. It is recommended such type of variables at class level by using static modifier.
- In case of static variables entire class will maintain only one copy and is shared by every object of that class.
- Static variables are created at the time of class loading and scope of the static variable is exactly same as the scope of the .class file.
- Static variables are stored in method area and are declared in the class but outside of any method, block or constructor.
- Static variables can be accessed from both instance and static areas directly.
- We can access static variables either by class name or by object reference but usage of class name is recommended.
- Static variables also known as class level variables or fields.

**Example**

```
class ref
{
        static int s=20;
        int i=5;
        boolean b;
        public static void main (String[] args)
        {
                //System.out.println(i);        //C.E:non-static variable i cannot be referenced from a //static
                                                context(invalid)
                ref r=new ref ();
                System.out.println(r.i);        // output 5(valid)
                System.out.println(r.b);
                r. func();
                System.out.println("static variable");
                System.out.println(r.s);
                System.out.println(ref.s);
                System.out.println(s);          // output false(valid)

        }
        public void func ()
        {
                System.out.println (i);         //output 5(valid)
                System.out.println (b);         //output false(valid)
        }
```

```
}
```

**Local variables**

- Variables declared inside a method or block or constructors are called local variables, temporary variables or stack variables.
- The local variables are created as part of the block it is declared and are destroyed once that block completes its execution.
- The scope of the local variables is same as scope of the block in which it is declared.
- The local variables are stored in the stack.
- The only applicable modifier for local variables is final

```
class Test
{
	public static void main (String[] args)
	{
		int i=0;
		for(int j=0;j<3;j++)
		{
			i=i+j;
		}
		System.out.println(i+"---"+j);
	}
}
```

**Arrays**

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. To declare an array, define the variable type with **square brackets**.

**Example**

```
int[] a;		//recommended to use because name is clearly separated from the type
int []b;
int c[];
```

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

**Example**

```
int[] a;		//valid
int[5] a;		//invalid
```

**Two-dimensional array declaration**

**Example**

```
int[][] a;
int [][]a;
int a[][];
int[] []a;
int[] a[];
int []a[];
```

**Three-dimensional array declaration**

**Example**

```
int[][][] a;
int [][][]a;
int a[][][];
int[] [][]a;
```

```
int[] a[][];
int[] []a[];
int[][] []a;
int[][] a[];
int []a[][];
int [][]a[];
```

**Which of the following declarations are valid?**

1) int[] al, bl; a1 and b1 one dimension
2) int[] a2[],b2; a2 is two-dimensional, b2 one dimensional
3) int[] []a3,b3; both two dimensional
4) int[] a, []b; //C.E:<identifier> expected(invalid)

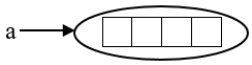
If we want to specify dimension before the variable that facility is applicable only for first variable in a declaration. If we try to apply for remaining variable, we will get compile time error.

**Array construction**

Every array in java is an object and new operator is used for creating it.

Example

```
int[] a=new int[3];
```




For every array type there is corresponding class. These classes are part of java language but not available at programmer level.

```
System.out.println(a.getClass().getName());   //it will print corresponding class of array.
```

| Array | Class |
|---|---|
| int[] | [I |
| int[][] | [[I |
| double[] | [D |
| boolean[] | [Z |
| byte[] | [B |
| short[] | [S |

**Rules of array creation**

- It is compulsory to specify the size of array at the time of creation otherwise we will get compile time error.
- Array with size zero is valid in java.

```
Class size
{
        public static void main (String [] args)
        {
                System.out.println(args.length)                 //output is Zero
        }
}
```

- Array size with -ve int value is invalid. If we take negative value then we will get runtime exception saying
  *NegativeArraySizeException.*
- Only byte, short, char, int data types are allowed to specify array size else we will get compile time error.
- The maximum allowed array size in java is maximum value of int size

**Example**

```
int[] a=new int[3];
int[] a=new int[];              //C.E:array dimension missing

int[] a=new int[0];             //valid

int[] a=new int[-3];            //R.E:NegativeArraySizeException

int[] a=new int['a'];           //(valid)

byte b=10;
int[] a=new int[b];             //(valid)
short s=20;
int[] a=new int[s];             //(valid)

int[] a=new int[10L];           //C.E:possible loss of precision (invalid)
int[] a=new int[l0.5];          //C.E:possible loss of precision (invalid)

int[] al=new int[2147483647];       (valid)
```

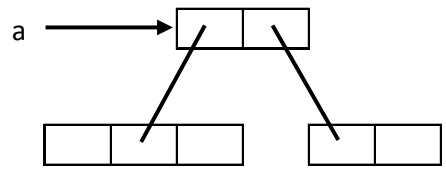
**Two-dimensional array creation**

In java multidimensional arrays are implemented as array of arrays approach but not matrix form.
The main advantage of this approach is to improve memory utilization.

**Example**

```
int[][] a=new int[2] [];
a[0]=new int[3];
a[l]=new int[2];
```

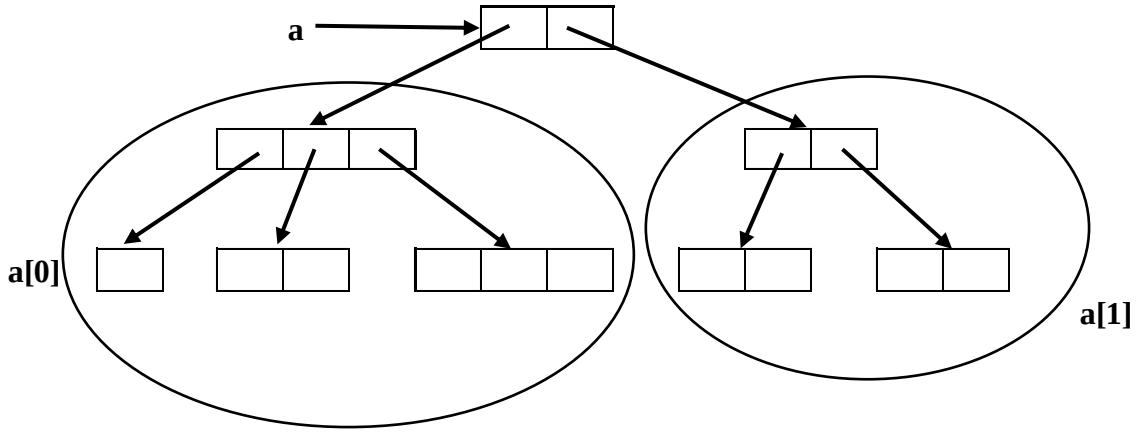
**Memory Representation**




**Example**

```
int[][][] a=new int[2][][];
a[0]=new int[3][];
a[0][0]=new int[l];
a[0][l]=new int[2];
a[0][2]=new int[3];
a[l]=new int[2][2];
```

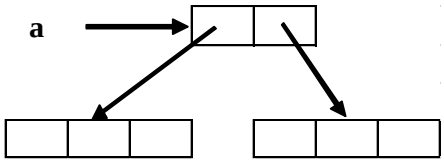
**Memory Representation**




**Array initialization**
Every element is initialized with default value automatically.
**Example**

```
int[] a=new int[3];
System.out.println(a);          //[1@3e25a5
System.out.println(a[0]);       //0
```

Whenever we are trying to print any object reference internally toString() method will be executed which is implemented by default to return the following.
classname@hexadecimalstringrepresentationofhashcode.

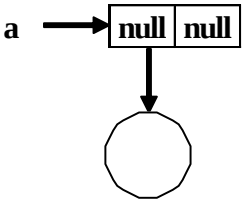



**Example:**

```
int [][] a=new int[2][3]
System.out.println (a);         //[[1@3e25a5
System.out.println(a [0]);      //[1@19821f
System.out.println(a [0] [0]);  //0
```

**Example**

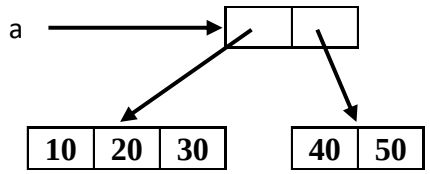
```
int[][] a=new int[2][];
System.out.println(a);          //[[I@3e25a5
System.out.println(a[0]);       //null
System.out.println(a[0][0]);    //R.E:NullPointerException
```




**Declaration construction and initialization of an array in a single line**
Arrays can also be initialized with customized values like:
**Example**

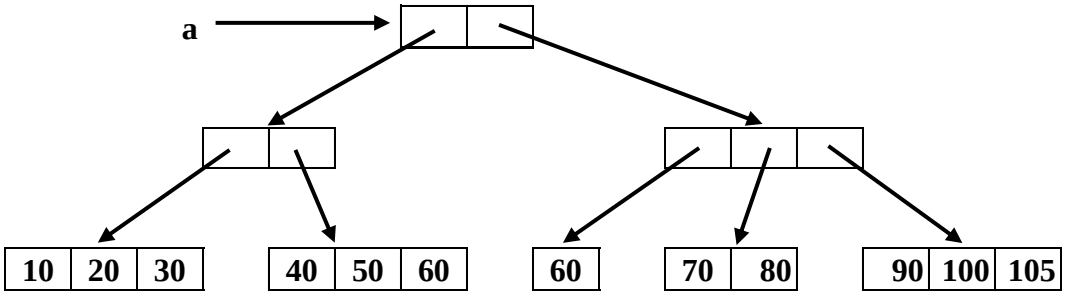
```
int[] a=new int[4];
a[0]=l0;
a[1]=20;
a[2]=30;
a[3]=40;  or int[] a={l0,20,30};

int[][] a={{l0,20,30},{40,50}};
```




Int [][][]={{{10,20,30},{40,50,60}},{{60},{70,80},{90,100,105}}}

**Memory Representation**




System.out.println(a[0][l][l]); //50(valid)
System.out.println( a[l] [ 0] [2]) ; //R. E:ArraylndexOutOfBoundsException: 2(invalid)
System.out.println(a[1][2][1]); //100(valid)
System.out.println{ a[ 1] [2] [2]); //l05(valid)
System.out.println{ a[2] [ 1] [ 0]) ; //R. E:ArraylndexOutOfBoundsException: 2(invalid)
System.out.println{a[l][l][l]); //80(valid)

**length Vs length()**
**length**
It is the final variable applicable only for arrays.
It represents the size of the array.

**Example:**
int[] x=new int[3];
System.out.println(x.length()); //C.E: cannot find symbol
System.out.println(x.length); //3

length() method
It is a final method applicable for String objects.
It returns the no of characters present in the String.

Example
String s="computer";
System.out.println(s.length); //C.E:cannot find symbol
System.out.println(s.length()); //8

For multidimensional arrays length variable will return only base size but not total size.
Example: int[][] a=new int[6][3];
System.out.println(a.length); //6
System.out.println(a[0].length); //3

**Anonymous Arrays**

An array without name is called anonymous arrays. The main objective of anonymous arrays is "just for instant use". We can create anonymous array as follows.

```
new int[]{10,20,30,40};          (valid)
new int[J[]{{10,20},{30,40}};    (valid)
```

At the time of anonymous array creation we can't specify the size otherwise we will get compile time error.

Example

```
class Anarray
{
        public static void main(String [] args)
        {
                System.out.println(sum(new int[]{1,2,3,4,5}));
         }
        public static int sum(int [] a)
        {
                int sum=0;
                for (int i :a)
                {
                        sum=sum+i;
                 }
                return sum;
         }
}
```

**Array element assignment**

**Case 1**

```
int [] x=new int[5]
x[0]=10;
x[1]='a';               //97 will be stored
byte b=30
x[2]=b;
short s=30
x[3]=s;
x[5]=10L                //CE: possible loss of precision found: long required: int.
```

In case of primitive type array awe can provide any type of element which can be promoted to declared type. For float type arrays the allowed element types are byte, short, char, int, long, float.

**Case 2:**

The case of Object type arrays as array elements we can provide either declared type objects or its child class objects.

```
Object[] a=new Object[10];
a[0]=new lnteger(10);                   //(valid)
a[1]=new Object();                      //(valid}
a[2]=new String("CSE"};                 //(valid)

Number[] n=new Number[10];
n[0]=new lnteger(10);                   //{valid)
n[l]=new Double(10.5);                  //{valid)
n[2]=new String("CSE");                 //C.E:incompatible types/ /(invalid)
```

**Case 3**
In the case of interface type arrays as array elements we can provide its implemented class objects.

```
Runnable[] r=new Runnable[10];
r[0]=new Thread();
r[l]=new String("CSE");                    //C.E: incompatible types
```