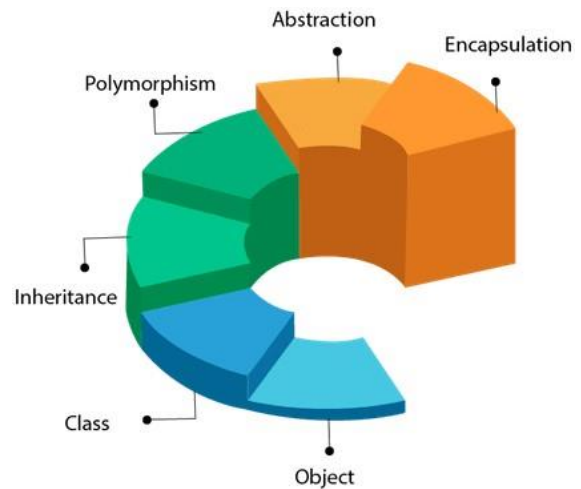


OOPs (OBJECT – ORIENTED PROGRAMMING SYSTEM) CONCEPT

o Introduction to OOP Concept

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- o Object
- o Class
- o Inheritance
- o Polymorphism
- o Abstraction
- o Encapsulation



Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- o Coupling
- o Cohesion
- o Association
- o Aggregation
- o Composition

o Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.



o Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

o Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

o Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.



o Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

o Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.



Capsule

o Coupling

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

o Cohesion

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The `java.io` package is a highly cohesive package because it has I/O related classes and interface. However, the `java.util` package is a weakly cohesive package because it has unrelated classes and interfaces.

o Association

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Let's understand the relationship with real-time examples. For example, One country can have one prime minister (one to one), and a prime minister can have many ministers (one to many). Also, many MP's can have one prime minister (many to one), and many ministers can have many departments (many to many).

Association can be unidirectional or bidirectional.

o Aggregation

Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state. It represents the weak relationship between objects. It is also termed as a *has-a* relationship in Java. Like, inheritance represents the *is-a* relationship. It is another way to reuse objects.

o Composition

The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state. There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.

Netzwerk Academy

o Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.
2. OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.
3. OOPs provides the ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

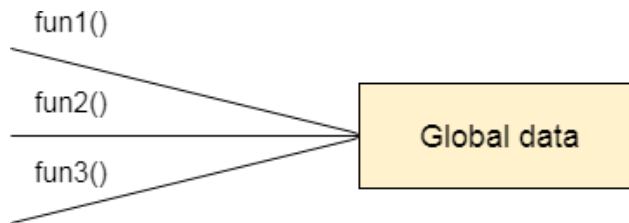


Figure: Data Representation in Procedure-Oriented Programming

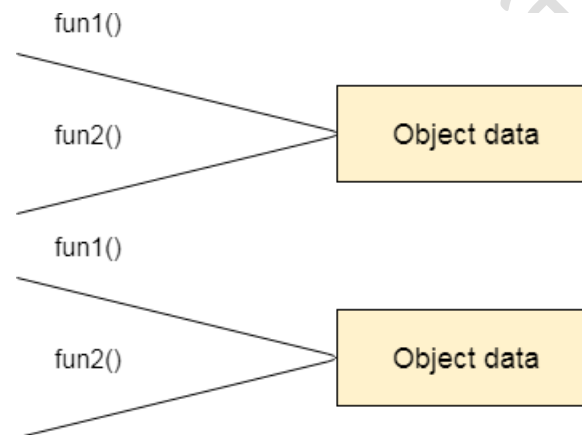


Figure: Data Representation in Object-Oriented Programming

What is the difference between an object-oriented programming language and object-based programming language?

Object-based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object-based programming languages.

DATA TYPES, VARIABLES AND ARRAYS

□ Java Identifiers

Identifiers in Java are symbolic names used for identification. They can be a class name, variable name, method name, package name, constant name, and more. However, In Java There are some reserved words that cannot be used as an identifier.

For every identifier there are some conventions that should be used before declaring them. Let's understand it with a simple Java program:

```
public class demo {
    public static void main(String[] args) { System.out.println("Hello
        Netzwerk Academy");
    }
}
```

From the above example, we have the following Java identifiers:

1. HelloJava (Class name)
2. main (main method)
3. String (Predefined Classname)
4. args (String variables)
5. System (Predefined class)
6. out (Variable name)
7. println (method)

Rules for Identifiers in Java

There are some rules and conventions for declaring the identifiers in Java. If the identifiers are not properly declared, we may get a compile-time error. Following are some rules and conventions for declaring identifiers:

- o A valid identifier must have characters [A-Z] or [a-z] or numbers [0-9], and underscore (_) or a dollar sign (\$). for example, @netzwerk is not a valid identifier because it contains a special character which is @.
- o There should not be any space in an identifier. For example, netz werk is an invalid identifier.
- o An identifier should not contain a number at the starting. For example, 123netzwerk is an invalid identifier.
- o An identifier should be of length 4-15 letters only. However, there is no limit on its length. But, it is good to follow the standard conventions.
- o We can't use the Java reserved keywords as an identifier such as int, float, double, char, etc. For example, int double is an invalid identifier in Java.
- o An identifier should not be any query language keywords such as SELECT, FROM, COUNT, DELETE, etc.

Java Reserved Keywords

Java reserved keywords are predefined words, which are reserved for any functionality or meaning. We cannot use these keywords as our identifier names, such as class name or method name. These keywords are used by the syntax of Java for some functionality. If we use a reserved word as our variable name, it will throw an error.

In Java, every reserved word has a unique meaning and functionality.

Consider the below syntax:

double marks;

in the above statement, double is a reserved word while marks is a valid identifier.

Below is the list of reserved keywords in Java:

abstract	continue	for	protected	transient
Assert	Default	Goto	public	Try
Boolean	Do	If	Static	throws
break	double	implements	strictfp	Package
byte	else	import	super	Private
case	enum	Interface	Short	switch
Catch	Extends	instanceof	return	void
Char	Final	Int	synchronized	volatile
class	finally	long	throw	Date
const	float	Native	This	while

Although the const and goto are not part of the Java language; But, they are also considered keywords.

Example of Valid and Invalid Identifiers

Following are some examples of valid identifiers in Java:

- TestVariable
- testvariable
- a
- i
- Test_Variable
- _testvariable
- \$testvariable
- sum_of_array
- TESTVARIABLE
- test123

Netzwerk Academy

Below are some examples of invalid identifiers:

- Test Variable (We cannot include a space in an identifier)
- 123test (The identifier should not begin with numbers)
- test+variable (The plus (+) symbol cannot be used)
- test-variable (Hyphen symbol is not allowed)
- test_&_variable (ampersand symbol is not allowed)
- Test'variable (we cannot use an apostrophe symbol in an identifier)

We should follow some naming convention while declaring an identifier. However, these conventions are not forced to follow by the Java programming language. That's why it is called conventions, not rules. But it is good to follow them. These are some industry standards and recommended by Java communities such as Oracle and Netscape.

If we do not follow these conventions, it may generate confusion or erroneous code.

□ Data Types

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.
2. Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

1. Java Primitive Data Types:

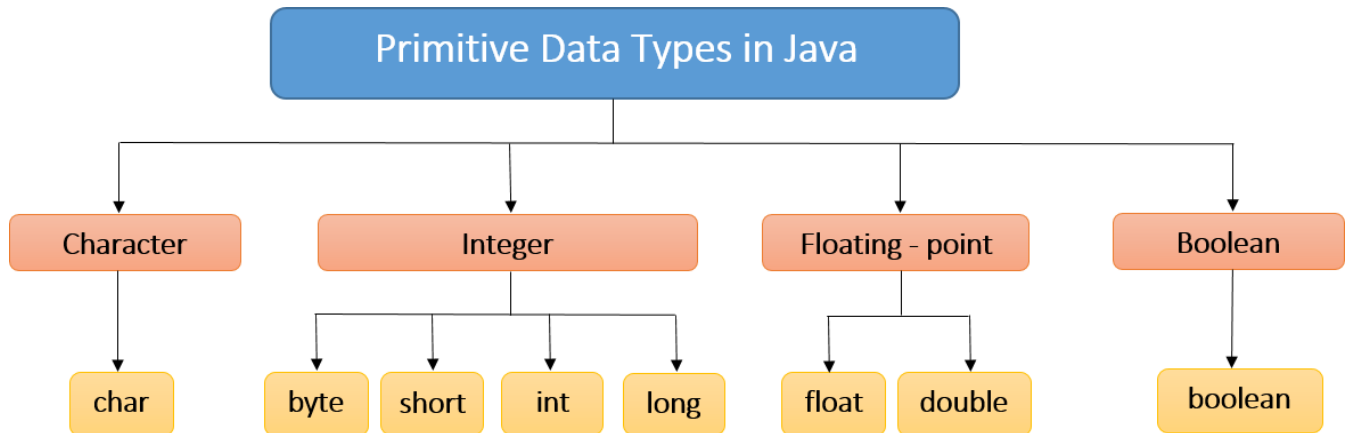
In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

Java is a statically-typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

- boolean data type
- byte datatype
- char data type
- short data type
- int data type

- long data type
- float data type
- double data type



Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example:

Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example:

byte a = 127, **byte** b = -128

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example:

short s = 10000, **short** r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example:

int a = 100000, **int** b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808 (-2^{63}) to 9,223,372,036,854,775,807 ($2^{63} - 1$) (inclusive). Its minimum value is -9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example:

long a = 100000L, **long** b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0f.

Example:

float f1 = 123.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example:

```
double d1 = 14.7
```

Netzwerk Academy

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example:

```
char letterN = 'N'
```

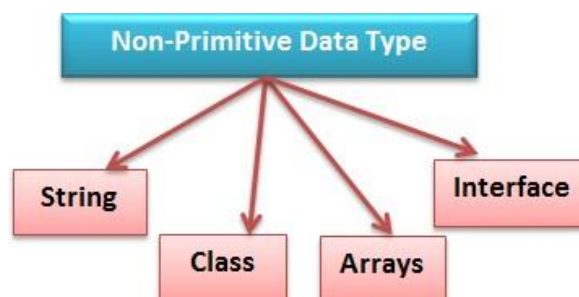
Why char uses 2 byte in java and what is \u0000 ?

It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system.

Data Type	Default Value	Default Size	Value Range	Example
boolean	false	1 bit (which is a special type for representing true/false values)	true/false	boolean b=true;
char	'\u0000'	2 byte (16 bit unsigned unicode character)	0 to 65,535	char c='a';
byte	0	1 byte (8 bit Integer data type)	-128 to 127	byte b=10;
short	0	2 byte (16 bit Integer data type)	-32768 to 32767	short s=11;
int	0	4 byte (32 bit Integer data type)	-2147483648 to 2147483647.	int i=10;
long	0L	8 byte (64 bit Integer data type)	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long l=100012;
float	0.0f	4 byte (32 bit float data type)	1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative).	float f=10.3f;
double	0.0d	8 byte (64 bit float data type)	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	double d=11.123;

2. Non-Primitive Datatypes

Non-Primitive datatypes refer to objects and hence they are called reference types. Examples of non-primitive types include Strings, Arrays, Classes, Interface, etc.



String Data Type

String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The *java.lang.String* class is used to create a string object.

Class Data Type

A class in Java is a blueprint which includes all your data. A class contains fields (variables) and methods to describe the behavior of an object.

Arrays Data Type

Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index.

Interface Data Type

Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

Difference between primitive and non-primitive data types

- Primitive types are predefined in Java. Non-primitive types are created by the programmer and are not defined by Java.
- Non Primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type always has a value, whereas non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types start with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

□ How to define our own Data type in Java (enum)

enum in Java

Enumerations serve the purpose of representing a group of named constants in a programming language.

The Enum in Java is a data type which contains a fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY), directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.

According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

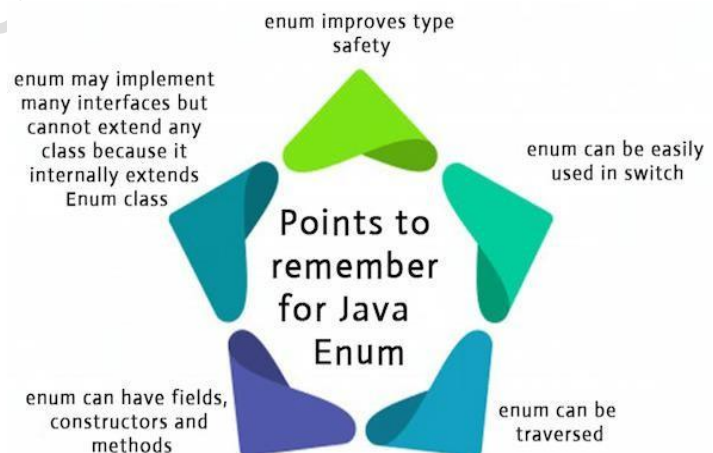
Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class or outside the class but not inside a Method

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java enum

- Enum improves typesafety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class



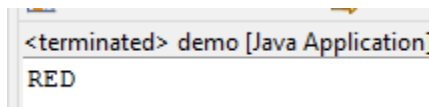
Simple Example of Java Enum

- A simple enum example where enum is declared outside any class (Note enum keyword instead of class keyword)

```
enum Color {
    RED,
    GREEN,
    BLUE;
}

public class demo {
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED; System.out.println(c1);
    }
}
```

Output:



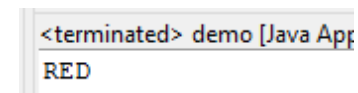
```
<terminated> demo [Java Application]
RED
```

- enum declaration inside a class.

```
public class demo {
    enum Color {
        RED,
        GREEN,
        BLUE;
    }

    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED; System.out.println(c1);
    }
}
```

Output:



```
<terminated> demo [Java App]
RED
```


- A Java program to demonstrate working on enum in switch case (Filename demo.java)

```
//An Enum class
enum Day {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY;
}

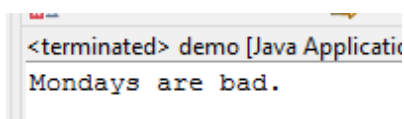
//Driver class that contains an object of "day" and
//main().
public class demo { Day
    day;

    // Constructor
    public demo(Day day) { this.day = day; }

    // Prints a line about Day using switch
    public void dayIsLike()
    {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break; case
            FRIDAY:
                System.out.println("Fridays are better.");
                break; case
            SATURDAY:
            case SUNDAY:
                System.out.println("Weekends are best.");
                break;
            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }

    // Driver method
    public static void main(String[] args)
    {
        String str = "MONDAY";
        demo t1 = new demo(Day.valueOf(str));
        t1.dayIsLike();
    }
}
```

Output:



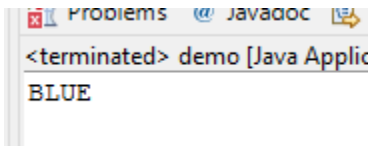
```
<terminated> demo [Java Applicati
Mondays are bad.
```

- A Java program to demonstrate that we can have main() inside enum class.

```
enum demo {
    RED,
    GREEN,
    BLUE;

    // Driver method
    public static void main(String[] args)
    {
        demo c1 = demo.BLUE; System.out.println(c1);
    }
}
```

Output:



enum and Inheritance:

- All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
- toString() method is overridden in java.lang.Enum class, which returns enum constant name.
- enum can implement many interfaces.

values(), ordinal() and valueOf() methods:

- These methods are present inside java.lang.Enum.
- values() method can be used to return all values present inside the enum.
- Order is important in enums. By using the ordinal() method, each enum constant index can be found, just like an array index.
- valueOf() method returns the enum constant of the specified string value if exists.

- Java program to demonstrate working of values(), ordinal() and valueOf()

```
enum Color {
    RED,
    GREEN,
    BLUE;
}

public class demo {
    public static void main(String[] args)
    {
        // Calling values()
        Color arr[] = Color.values();

        // enum with loop
        for (Color col : arr) {
            // Calling ordinal() to find index
            // of color.
            System.out.println(col + " at index "
                               + col.ordinal());
        }

        // Using valueOf(). Returns an object of
        // Color with given constant.
        // Uncommenting second line causes exception
        // IllegalArgumentException System.out.println(Color.valueOf("RED"));
        // System.out.println(Color.valueOf("WHITE"));
    }
}
```

Output:

```
<terminated> demo [Java Application]
RED at index 0
GREEN at index 1
BLUE at index 2
RED
```

enum and constructor:

- enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

enum and methods:

enum can contain both concrete methods and abstract methods. If an enum class has an abstract method, then each instance of the enum class must implement it.

Netzwerk Academy

- Java program to demonstrate that enums can have constructor and concrete methods.

//An enum (Note enum keyword inplace of class keyword)

```
enum Color {
    RED,
    GREEN,
    BLUE;

    // enum constructor called separately for each
    // constant
    private Color()
    {
        System.out.println("Constructor called for : "
                           + this.toString());
    }

    public void colorInfo()
    {
        System.out.println("Universal Color");
    }
}

public class demo {
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED; System.out.println(c1);
        c1.colorInfo();
    }
}
```

Output:

```
<terminated> demo [Java Application] C:\Program
Constructor called for : RED
Constructor called for : GREEN
Constructor called for : BLUE
RED
Universal Color
```

Difference between enums and Classes

An enum can, just like a class, have attributes and methods. The only difference is that enum constants are public, static and final (unchangeable - cannot be overridden).

An enum cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why and when to use enums?

Use enums when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Netzwerk Academy

□ Literals in Java

Any constant value which can be assigned to the variable is called literal/constant.

In simple words, Literals in Java is a synthetic representation of boolean, numeric, character, or string data. It is a medium of expressing particular values in the program, such as an integer variable named 'count' is assigned an integer value in the following statement.

```
//Here 100 is a constant/literal. int
x = 100;
```

Integral literals

For Integral data types (byte, short, int, long), we can specify literals in 4 ways:-

- Decimal literals (Base 10): In this form, the allowed digits are 0-9.

```
int x = 101;
```

- Octal literals (Base 8): In this form, the allowed digits are 0-7.

```
//The octal number should be prefix with 0.
int x = 0146;
```

- Hexa-decimal literals (Base 16): In this form, the allowed digits are 0-9, and characters are a-f. We can use both uppercase and lowercase characters as we know that java is a case-sensitive programming language, but here java is not case-sensitive.

```
//The hexa-decimal number should be prefix
//with 0X or 0x.
int x = 0X123Face;
```

- Binary literals: From 1.7 onward, we can specify literal value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.

```
int x = 0b1111;
```

- Java program to illustrate the application of Integer literals

```
public class demo {  
    public static void main(String[] args)  
    {  
        // decimal-form literal  
        int a = 101;  
        // octal-form literal  
        int b = 0100;  
        // Hexa-decimal form literal  
        int c = 0xFace;  
        // Binary literal  
        int d = 0b1111;  
    }  
}
```

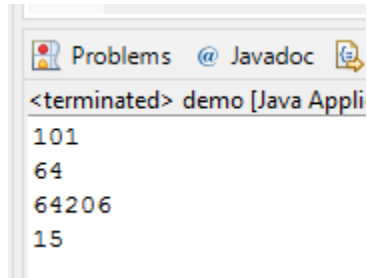
Netzwerk Academy


```

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
    }
}

```

Output:



```

<terminated> demo [Java Appli
101
64
64206
15

```

Note: By default, every literal is of int type, we can specify explicitly as long type by suffixed with l or L. There is no way to specify byte and short literals explicitly but indirectly we can specify. Whenever we are assigning integral literal to the byte variable and if the value is within the range of byte then the compiler treats it automatically as byte literals.

Floating-Point literal

For Floating-point data types, we can specify literals in only decimal form, and we can't specify in octal and Hexadecimal forms.

- o Decimal literals (Base 10): In this form, the allowed digits are 0-9.

```
double d = 123.456;
```

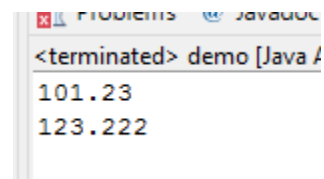
Java program to illustrate the application of floating-point literals

```

public class demo {
    public static void main(String[] args)
    {
        // decimal-form literal
        double a = 101.230;
        // It also acts as decimal literal
        double b = 0123.222;
        // Hexa-decimal form (error)
        //double c = 0x123.222;

        System.out.println(a);
        System.out.println(b);
        // System.out.println(c);
    }
}

```



```

<terminated> demo [Java /
101.23
123.222

```

Note: By default, every floating-point 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 f or F. We can specify explicitly floating-point literal as double type by suffixed with d or D. Of course this convention is not required.

Char literals

For char data types, we can specify literals in 4 ways:

- Single quote: We can specify literal to a char data type as a single character within the single quote.

```
char ch = 'a';
```

- Char literal as Integral literal: we can specify char literal as integral literal, which represents the Unicode value of the character, and that integral literal can be specified either in Decimal, Octal, and Hexadecimal forms. But the allowed range is 0 to 65535.

```
char ch = 062;
```

- Unicode Representation: We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.

```
char ch = '\u0061'; // Here /u0061 represent a.
```

- Escape Sequence: Every escape character can be specified as char literals.

```
char ch = '\n';
```

□ Java program to illustrate the application of char literals

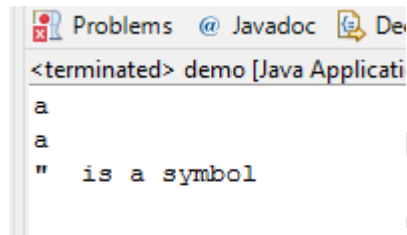
```

public class demo {
    public static void main(String[] args)
    {
        // single character literal within single quote
        char ch = 'a';
        // It is an Integer literal with octal form
        //char b = 0789; //Integer number too large
        // Unicode representation
        char c = '\u0061';

        System.out.println(ch);
        // System.out.println(b); //Output: Error: Integer number too large
        System.out.println(c);

        // Escape character literal
        System.out.println("\" is a symbol");
    }
}

```



String literals

Any sequence of characters within double quotes is treated as String literals.

```
String s = "Hello";
```

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile-time expressions, so the following String expression results in a string with three lines of text:

Example:

```
String text = "This is a String literal\n"
    + "which spans not one and not two\n"
    + "but three lines of text.\n";
```

- Java program to illustrate the application of String literals

```
public class demo {
    public static void main(String[] args)
    {
        String s = "Hello";

        // If we assign without "" then it treats
        // as a variable and causes compiler error String s1 = Hello;

        System.out.println(s); System.out.println(s1);
    }
}
```

Boolean literals

Only two values are allowed for Boolean literals, i.e., true and false.

```
boolean b = true;
```

- Java program to illustrate the application of boolean literals

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

```
{  
  boolean b = true; boolean  
  c = false; boolean d = 0;  
  boolean b = 1;  
  
  System.out.println(b); System.out.println(c);  
  System.out.println(d); //error: incompatible types: int cannot be converted to boolean  
  System.out.println(e); //error: incompatible types: int cannot be  
converted to boolean  
}  
}
```

Netzwerk Academy

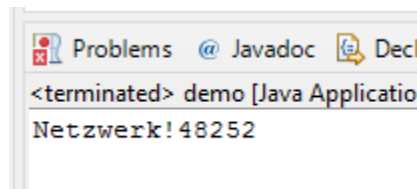
Note: When we are performing concatenation operations, then the values in brackets are concatenated first. Then the values are concatenated from the left to the right. We should be careful when we are mixing character literals and integers in String concatenation operations and this type of operation are known as Mixed Mode operation.

```
//Java program to illustrate the behaviour of
//char literals and integer literals when
//we are performing addition

public class demo {
    public static void main(String[] args)
    {
        // ASCII value of 0 is 48
        int first = '0';

        // ASCII value of 4 is 52
        int second = '4'; System.out.println("Netzwerk!" + first +
                                            '2' + second);
    }
}
```

Output:



Explanation: Whenever we are performing addition between a string and integer, the overall result is converted into a string. The above program evaluation is done in the following way:

"Netzwerk!" + first + '2' + second

"Netzwerk!" + 48 + '2' + 52

"Netzwerk!48" + '2' + 52

"Netzwerk!482" + 52

"Netzwerk!48252"

Why use literals?

To avoid defining the constant somewhere and making up a label for it. Instead, to write the value of a constant operand as a part of the instruction.

□ Variable & Declarations of Variable

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

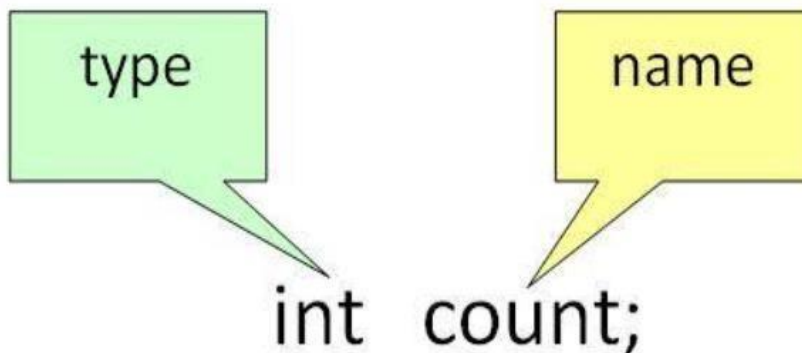
A variable is a name given to a memory location. It is the basic unit of storage in a program

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed.

How to Declare Variable?

We can declare variables in java as pictorially depicted below as a visual aid.



From the image, it can be easily perceived that while declaring a variable, we need to take care of two things that are:

1. Datatype: Type of data that can be stored in this variable
2. Dataname: Name was given to the variable.

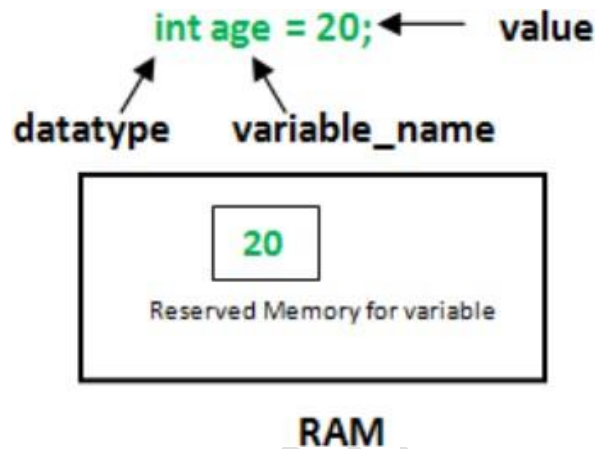
In this way, a name can only be given to a memory location. It can be assigned values in two ways:

1. Variable Initialization
2. Assigning value by taking input

How to initialize variables?

It can be perceived with the help of 3 components that are as follows:

- datatype: Type of data that can be stored in this variable.
- variable_name: Name given to the variable.
- value: It is the initial value stored in the variable.



Types of Variables in Java

Now let us discuss different types of variables which are listed as follows:

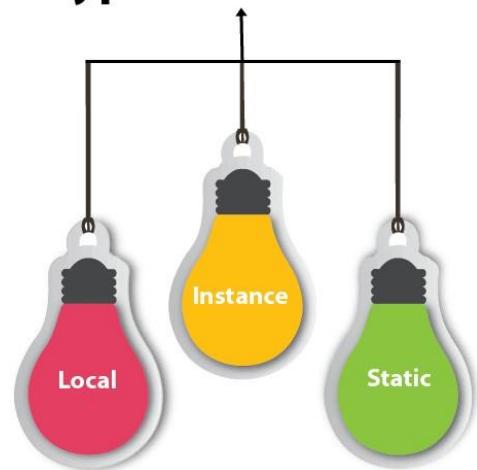
1. Local Variables
2. Instance Variables
3. Static Variables

Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

Types of Variables



Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

Static Variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

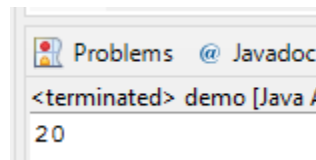
Example to understand the types of variables in java

```
public class demo
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
} //end of class
```

Java Variable Example: Add Two Numbers

```
public class demo
{
    public static void main(String[] args){
        int a=10; int
        b=10; int
        c=a+b;
        System.out.println(c);
    }
}
```

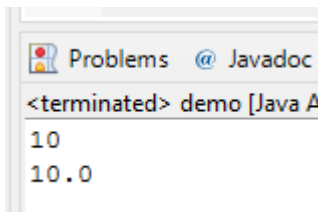
Output:



Java Variable Example: Widening

```
public class demo
{
    public static void main(String[] args){
        int a=10;
        float f=a; System.out.println(a);
        System.out.println(f);
    }
}
```

Output:

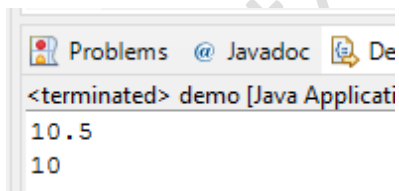


```
<terminated> demo [Java A
10
10.0
```

Java Variable Example: Narrowing (Typecasting)

```
public class demo
{
    public static void main(String[] args){
        float f=10.5f;
        //int a=f; //Compile time error
        int a=(int)f;
        System.out.println(f); System.out.println(a);
    }
}
```

Output:



```
<terminated> demo [Java Applicati
10.5
10
```

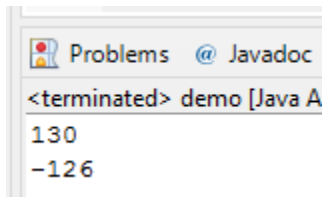
Java Variable Example: Overflow

```

class demo
{
    public static void main(String[] args){
        //Overflow
        int a=130;
        byte b=(byte)a;
        System.out.println(a);
        System.out.println(b);
    }
}

```

Output:



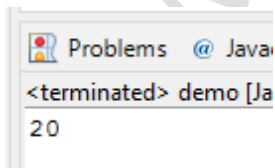
Java Variable Example: Adding Lower Type

```

class demo
{
    public static void main(String[] args){
        byte a=10;
        byte b=10;
        //byte c=a+b; //Compile Time Error: because a+b=20 will be int
        byte c=(byte)(a+b); System.out.println(c);
    }
}

```

Output:



□ Scope of Variables

In programming, scope of variable defines how a specific variable is accessible within the program or across classes.

In programming, a variable can be declared and defined inside a class, method, or block. It defines the scope of the variable i.e. the visibility or accessibility of a variable. Variable declared inside a block or method are not visible to outside. If we try to do so, we will get a compilation error. Note that the scope of a variable can be nested.

- We can declare variables anywhere in the program but it has limited scope.
- A variable can be a parameter of a method or constructor.
- A variable can be defined and declared inside the body of a method and constructor.
- It can also be defined inside blocks and loops.
- Variable declared inside main() function cannot be accessed outside the main() function

Variable Type	Scope	Lifetime
Instance Variable	Throughout the class except in static methods	Until the object is available in the memory
Class Variable	Throughout the class	Until the end of program
Local Variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Example of variables

```
public class demo
{
    //instance variable String name =
    "Naruto";
    //class and static variable
    static double height= 5.9;
    public static void main(String args[])
    {
        //local variable
        int marks = 72;
    }
}
```

Java scope rules can be covered under following categories.

- Member Variables (Class Level Scope)
- Local Variables (Method Level Scope)

Member Variables (Class Level Scope)

These are the variables that are declared inside the class but outside any function have class-level scope. We can access these variables anywhere inside the class. Note that the access specifier of a member variable does not affect the scope within the class. Java allows us to access member variables outside the class with the following rules:

Access Modifier	Package	Subclass	World
public	Yes	Yes	Yes
protected	Yes	Yes	No
private	No	No	No
default	Yes	No	No

Syntax

```

public class demo
{
    //variables declared inside the class have class level scope
    int age;
    private String name;
    void displayName()
    {
        //statements
    }
    int dispalyAge()
    {
        //statements
    }
    char c;
}

```

Let's see an example.

```

public class VariableScopeExample1
{
    public static void main(String args[])
    {
        int x=10;
        {
            //y has limited scope to this block only
            int y=20;
            System.out.println("Sum of x+y = " + (x+y));
        }
        //here y is unknown y=100;
        //x is still known x=50; } }

```

Output:

```
/VariableScopeExample1.java:12: error: cannot find symbol
y=100;
^
  symbol:   variable y
  location: class VariableScopeExample1
1 error
```

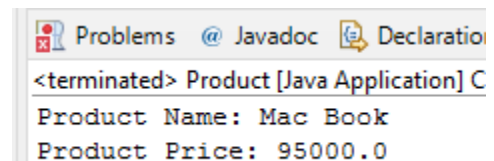
We see that `y=100` is unknown. If you want to compile and run the above program remove or comment the statement `y=100`. After removing the statement, the above program runs successfully and shows the following output.

```
Sum of x+y = 30
```

There is another variable named an instance variable. These are declared inside a class but outside any method, constructor, or block. When an instance variable is declared using the keyword `static` is known as a static variable. Their scope is class level but visible to the method, constructor, or block that is defined inside the class.

Example.

```
public class Product
{
    //variable visible to any child class
    public String pName;
    //variable visible to product class only
    private double pPrice;
    //creating a constructor and parsed product name as a parameter
    public Product (String pname)
    {
        pName = pname;
    }
    //function sets the product price
    public void setPrice(double pprice)
    {
        pPrice= pprice;
    }
    //method prints all product info
    public void getInfo()
    {
        System.out.println("Product Name: " +pName );
        System.out.println("Product Price: " +pPrice);
    }
    public static void main(String args[])
    {
        Product pro = new Product("Mac Book");
        pro.setPrice(95000);
        pro.getInfo();
    }
}
```

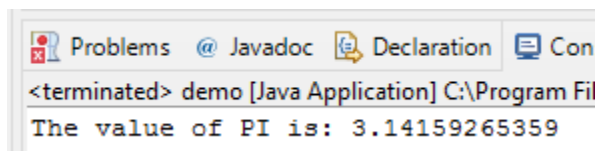


```
<terminated> Product [Java Application] C
Product Name: Mac Book
Product Price: 95000.0
```

Another Example for pi-value

```
public class demo
{
    //declaring a private static variable
    private static double pivalue;
    //declaring a constant variable
    public static final String piconstant = "PI";
    public static void main(String args[])
    {
        pivalue = 3.14159265359;
        System.out.println("The value of " + piconstant + " is: " + pivalue);
    }
}
```

Output:



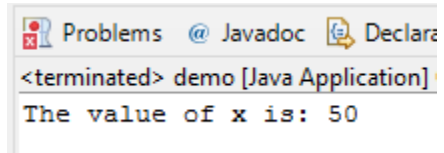
Local Variables (Method Level Scope)

These are the variables that are declared inside a method, constructor, or block have a method- level or block-level scope and cannot be accessed outside in which it is defined. Variables declared inside a pair of curly braces { } have block-level scope.

Program for Declaring a Variable Inside a Method

```
public class demo
{
    void show()
    {
        //variable declared inside a method has method level scope
        int x=50;
        System.out.println("The value of x is: "+x);
    }
    public static void main(String args[])
    {
        demo dc = new demo();
        dc.show();
    }
}
```

Output:



The screenshot shows a console window from an IDE. The title bar includes icons for 'Problems', 'Javadoc', and 'Declarations'. The main text area displays the output of a Java application named 'demo'. The output consists of two lines: the first line is '<terminated> demo [Java Application]' and the second line is 'The value of x is: 50'.

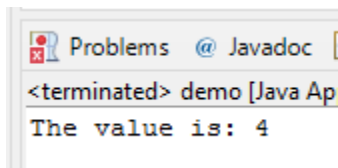
```
<terminated> demo [Java Application]  
The value of x is: 50
```

Netzwerk Academy

Let's see another example of method-level scope.

```
public class demo
{
    private int a;
    public void setNumber(int a)
    {
        this.a = a;
        System.out.println("The value is: "+a);
    }
    public static void main(String args[])
    {
        demo dc = new demo();
        dc.setNumber(4);
    }
}
```

Output:

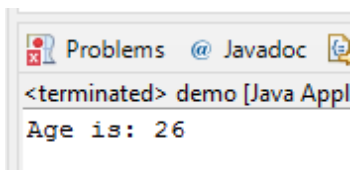


In the above example, we have passed a variable as a parameter. We have used *this* keyword that differentiates the class variable and local variable.

Program for Declaring a Variable Inside a Constructor

```
public class demo
{
    //creating a default constructor demo()
    {
        int age=26;
        System.out.println("Age is: "+age);
    }
    //main() method
    public static void main(String args[])
    {
        //calling a default constructor demo vc=new
        demo();
    }
}
```

Output:

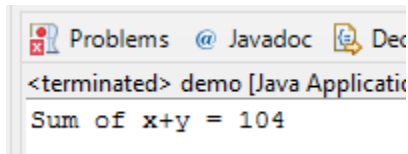


Program for Declaring a Variable Inside a Block

```
public class demo
{
    public static void main(String args[])
    {
        int x=4;
        {
            //y has limited scope to this block only
            int y=100;
            System.out.println("Sum of x+y = " + (x+y)); y=10;

            int y=150; //gives error, already defined
        }
        //creates a new variable
        int y;
    }
}
```

We see that y=150 is unknown. If you want to compile and run the above program remove or comment the statement y=150. After removing the statement, the above program runs successfully and shows the following output



Let's see another example for Block Scope with Loop

```
public class demo
{
    public static void main(String args[])
    {
        for (int x = 0; x < 10; x++)
        {
            System.out.println(x);
        }
        System.out.println(x);
    }
}
```

When we run the above program, it shows an error, cannot find symbol because we have tried to print the variable x that is declared inside the loop. To resolve this error, we need to declare the variable x just before the for loop.

```
public class demo
{
    public static void main(String args[])
    {
        int x;
        for (x=0; x<5; x++)
```

Netzwerk Academy

```

{
//prints 0 to 4
System.out.print(x+"\t"); //"\t" is to provide the tab space
}
//prints 5 System.out.println(x);
}
}

```

Output:

```

<terminated> demo [Java Application] C:\Program Files\Java\jdk-14.0.1\bin
0      1      2      3      4      5

```

□ Final Variable

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

Java Final Variable

If you make any variable as final, you cannot change the value of final variable (It will be constant). Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

Program to illustrate the Final Key work on Speed Limit and overriding the value

```

class demo{
    final int speedlimit=70; //final variable
    void run(){

```

```
speedlimit=100; //Compile time error bcoz already the speedlimit
                //is set to 70 by using final keyword
}
public static void main(String args[]){ demo obj=new
                demo();
obj.run();
}
    }//end of class
```

Netzwerk Academy

Java final method

If you make any method as final, you cannot override it. Program to illustrate the Final method and overriding the method

```
class demo{
    final void run(){System.out.println("running");}
}

class RE extends demo{
    void run(){System.out.println("running safely with 100kmph");
    //Compile error as its trying to override the method
}

public static void main(String args[]){ RE REBike= new
RE();
REBike.run();
}
}
```

Java final class

If you make any class as final, you cannot extend it.

Program to illustrate the Final class and extending the class **final**

```
class demo{
}

class RE extends demo{ //Compile error as its trying to extend the
//Final Class
    void run(){System.out.println("running safely with 100kmph");}

    public static void main(String args[]){ RE REBike= new
RE();
REBike.run();
}
}
```

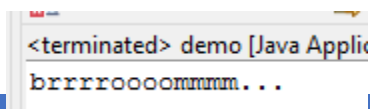
Is final method inherited?

Yes, final method is inherited but you cannot override it. For Example:

```
class demo{
    final void run(){System.out.println("brrrroooooommm...");
}
}

class RE2 extends demo{
    public static void main(String args[]){
        new RE2().run();
    }
}
```

Output:



What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}
```

Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
class Bike10{
    final int speedlimit; //blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}
```

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
class demo{
    static final int data; //static blank final variable
    static{ data=10;}
    public static void main(String args[]){
        System.out.println(demo.data);
    }
}
```

Can we declare a constructor final?

No, because constructor is never inherited.

□ Type Conversion and Casting

Type Casting

In Java, type casting is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

Syntax for Manual Type Cast:

destination_datatype = (target_datatype)variable; () is

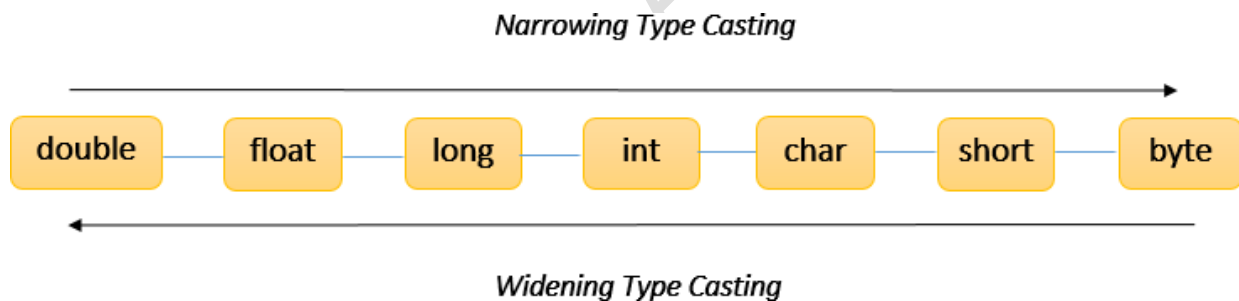
a casting operator

target_datatype: is a data type in which we want to convert the source data type.

Types of Type Casting

There are two types of type casting:

- Widening TypeCasting
- Narrowing TypeCasting



Widening Type Casting

Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion or casting down. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.

byte -> short -> char -> int -> long -> float -> double

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other

Netzwerk Academy

Example of Widening Type Casting

```
public class demo {

    public static void main(String[] args) {
        byte b = 77; short
        s = b; int i = s;
        long l = s; float f =
        s; double d = s;
        System.out.println("Examples of Widening Type casting...!!"); System.out.println("byte to
        short : "+s); System.out.println("byte to int : "+i); System.out.println("byte to long : "+l);
        System.out.println("byte to float : "+f); System.out.println("byte to double : "+d);
    }
}
```

Output:

```
<terminated> demo [Java Application] C:\Program Files\Java
Examples of Widening Type casting...!!
byte to short : 77
byte to int : 77
byte to long : 77
byte to float : 77.0
byte to double : 77.0
```

Narrowing Type Casting

Converting a higher data type into a lower one is called narrowing type casting. It is also known as explicit conversion or casting up. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

double -> float -> long -> int -> char -> short -> byte

Example of narrowing type casting

```
public class demo
{
    public static void main(String args[])
    {
        double d = 199.66;
        //converting double data type into long data type
        long l = (long)d;
    }
}
```

```
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
```

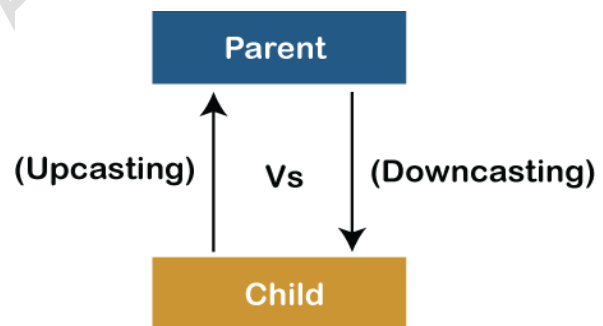
Output:

```
<terminated> demo [Java Application] C:\Program Files\Java
Before conversion: 199.66
After conversion into long type: 199
After conversion into int type: 199
```

In the above example, we have performed the narrowing typecasting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

Upcasting and Downcasting in Java

A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting. In Java, the object can also be typecasted like the data types. Parent and Child objects are two types of objects. So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting



Typecasting is used to ensure whether variables are correctly processed by a function or not. In Upcasting and Downcasting, we typecast a child object to a parent object and a parent object to a child object simultaneously. We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.

Upcasting

Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. Upcasting is also known as Generalization and Widening.

Netzwerk Academy

Objects of a class can be cast into objects of another class if both the classes are related to each other through the property of inheritance, i.e., one class is the parent class, and the other class is the child class.

This type of casting superclass object (parent class) will hold the sub-class object's properties. Let's understand upcasting using an example:

Animal.java (Parent class)

```
public class Animal {
    protected String name;
    protected int age;
    public Animal(String name, int age){
        this.name = name;
        this.age = age;
    }
    public void animalInfo() { System.out.println("Animal class info:");
        System.out.println("Name: "+this.name);
        System.out.println("Age: "+this.age);
    }
}
```

Dog.java (Child class)

```
public class Dog extends Animal {
    public String color;
    public Dog(String name, int age, String color){
        super(name, age);
        this.color = color;
    }
    public void dogInfo() { System.out.println("Dog class: ");
        System.out.println("Name: "+this.name);
        System.out.println("Age: "+this.age);
        System.out.println("Color: "+this.color);
    }
    public static void main(String[] args) { Dog dog = new
        Dog("Leo", 2, "Brown");
        Animal animal = new Animal("Casper", 3);
        animal = dog; //implicit casting Object of dog to Animal animal.animalInfo();
    }
}
```

Output:

```
<terminated> Dog [Java Applicat
Animal class info:
Name: Leo
Age: 2
```

In the above code, Animal class is called parent class, and Dog class is called child class because the Dog class extends the Animal class, and the Dog class has acquired all the properties of the Animal class:

- In the main() method, first, we have created an object of the Dog class using a new keyword followed by the creation of Animal class Object.
- In the second step, we have simply assigned the reference object of the Dog class to the animal class, i.e., animal=dog; this type of casting is known as implicit casting or widening or upcasting of objects.
- Widening takes place when a subclass object reference is assigned to a wider superclass object. Like, in the above example dog object was assigned to Animal object.

Another example for upcasting:

demo.java (Parent class)

```
class demo{
    void PrintData() {
        System.out.println("method of parent class");
    }
}
```

UpcastingExample.java (Child class)

```
class Child extends demo {
    public void PrintData() { System.out.println("method of child class");
    }
}
class UpcastingExample{
    public static void main(String args[]) {

        demo obj1 = (demo) new Child(); demo obj2
        = (demo) new Child(); obj1.PrintData();
        obj2.PrintData();

    } }
```

Output

```
<terminated> UpcastingExample [Java Applicat
method of child class
method of child class
```

Downcasting

Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Let's understand downcasting using an example:

Animal.java (Parent class)

```
class Animal{
    protected String name;
    protected int age;
    public Animal(String name, int age){
        this.name = name;
        this.age = age;
    }
    public void animalInfo() { System.out.println("Animal class info:");
        System.out.println("Name: "+this.name);
        System.out.println("Age: "+this.age);
    }
}
```

Dog.java (Child class)

```
public class Dog extends Animal {
    public String color;
    public Dog(String name, int age, String color){
        super(name, age);
        this.color = color;
    }
    public void dogInfo() { System.out.println("Dog class: ");
        System.out.println("Name: "+this.name);
        System.out.println("Age: "+this.age);
        System.out.println("Color: "+this.color);
    }
    public static void main(String[] args) { Animal animal = new
        Dog("Leo", 2, "Black");
        Dog dog = (Dog) animal; //implicit casting Object of student to
        person
        dog.animalInfo(); dog.dogInfo();
    }
}
```


Output:

```
<terminated> Dog [Java Applic
Animal class info:
Name: Leo
Age: 2
Dog class:
Name: Leo
Age: 2
Color: Black
```

In the above code, Animal class is the parent class, and Dog class is the child class because the Dog class extends Animal class, and the Dog class has acquired all the properties of the Animal class:

- In the main() method, first, we have created an object of the Dog class using the reference of the parent class, i.e., `Animal animal = new Dog("Leo", 2, "Black")`; otherwise, we'll encounter a runtime exception.
- In the second step, we have simply assigned the reference object of the Dog class to the animal class, i.e., `Dog dog = (Dog) animal`; this type of casting is known as explicit casting or narrowing or down-casting of objects.
- Narrowing typecasting occurs when a superclass object reference is narrow casted and assigned to a narrower sub-class object. Like, in the above example, an animal object was assigned to a Dog object reference.

Another example for downcasting in which both the valid and the invalid scenarios are explained Parent.java

(Parent class)

```
class Parent { String
    name;

    // A method which prints the data of the parent class
    void showMessage()
    {
        System.out.println("Parent method is called");
    }
}
```

Child.java (Child class)

```
class Child extends Parent {
    int age;

    // Performing overriding @Override
    void showMessage()
    {
        System.out.println("Child method is called");
    }
}
```

Netzwerk Academy

Downcasting.java

```
public class Downcasting{

    public static void main(String[] args)
    {
        Parent p = new Child(); p.name =
        "Naruto";

        // Performing Downcasting Implicitly
        //Child c = new Parent(); // it gives compile-time error

        // Performing Downcasting Explicitly Child c =
        (Child)p;

        c.age = 18; System.out.println(c.name);
        System.out.println(c.age); c.showMessage();

    }
}
```

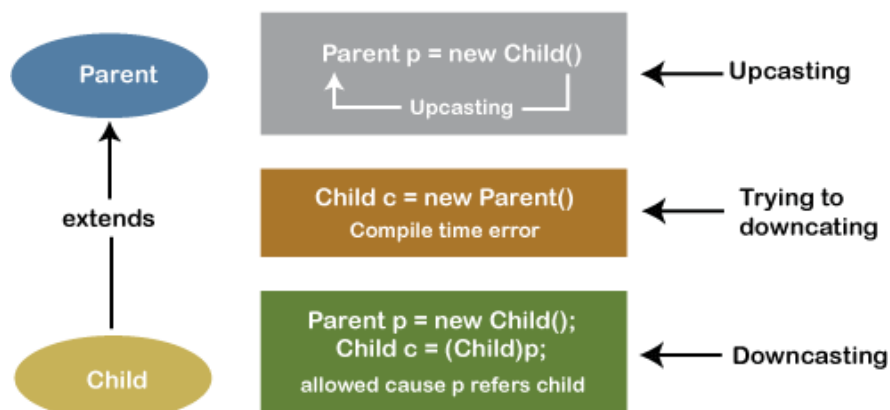
Output:

```
Naruto
18
Child method is called
```

Why we need Upcasting and Downcasting?

In Java, we rarely use Upcasting. We use it when we need to develop a code that deals with only the parent class. Downcasting is used when we need to develop a code that accesses behaviors of the child class.

Simply Upcasting and Downcasting



Difference between Upcasting and Downcasting

S.No	Upcasting	Downcasting
1.	A child object is typecasted to a parent object.	The reference of the parent class object is passed to the child class.
2.	We can perform Upcasting implicitly or explicitly.	Implicitly Downcasting is not possible.
3.	In the child class, we can access the methods and variables of the parent class.	The methods and variables of both the classes (parent and child) can be accessed.
4.	We can access some specified methods of the child class.	All the methods and variables of both classes can be accessed by performing downcasting.
5.	Parent p = new Parent()	Parent p = new Child() Child c = (Child)p;

Now we know about the concept of Widening and Narrowing, Upcasting and Downcasting, so now let us understand

The difference between Type Casting and Type Conversion

S.N.	Type Casting	Type Conversion
1	Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer.	Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code.
2	It can be used both compatible data type and incompatible data type.	Type conversion is only used with compatible data types, and hence it does not require any casting operator.
3	It requires a programmer to manually casting one data into another type.	It does not require any programmer intervention to convert one data type to another because the compiler automatically compiles it at the run time of a program.
4	It is used while designing a program by the programmer.	It is used or take place at the compile time of a program.
5	When casting one data type to another, the destination data type must be smaller than the source data.	When converting one data type to another, the destination type should be greater than the source data type.

6	It is also known as narrowing conversion because one larger data type converts to a smaller data type.	It is also known as widening conversion because one smaller data type converts to a larger data type.
---	--	---

Netzwerk Academy

7	It is more reliable and efficient.	It is less efficient and less reliable.
8	There is a possibility of data or information being lost in typecasting.	In type conversion, data is unlikely to be lost when converting from a small to a large data type.
9	<pre>float b=3.0; int a=(int) b</pre>	<pre>int x = 5, y = 2, c; float q= 12.5, p; p =q/x;</pre>

□ Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

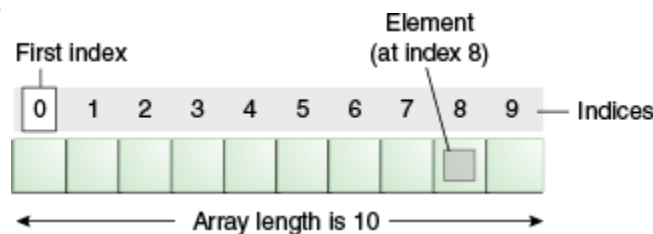
Java array is an object which contains elements of a similar datatype. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- Code Optimization: It makes the code optimized, we can retrieve or sort the data

efficiently.

- Random access: We can get any data located at an index position.

Netzwerk Academy

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or) dataType
[]arr; (or) dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Program to illustrate how to declare, instantiate, initialize and traverse the Java array

```
class demo{
    public static void main(String args[]){
        int a[]=new int[5]; //declaration and instantiation a[0]=10; //initialization
        a[1]=30;
        a[2]=20;
        a[3]=50;
        a[4]=40;
        //traversing array
        for(int i=0;i<a.length;i++) //length is the property of array System.out.println(a[i]);
    }
}
```

Output:

```
<terminated> demo [Ja
10
30
20
50
40
```


Declaration, Instantiation and Initialization of Java Array

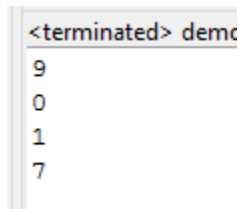
We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Program to illustrate the use of declaration, instantiation and initialization of Java array in a single line

```
class demo{
public static void main(String args[]){
int a[]={9,0,1,7};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array System.out.println(a[i]);
}
}
```

Output:



```
<terminated> demo
9
0
1
7
```

For-each Loop for Java Array

We can also print the Java array using for-each loop

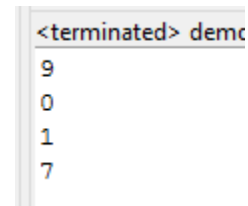
The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
    //body of the loop
}
```

Program to print the array elements using for-each loop

```
class demo{
public static void main(String args[]){
int arr[]={9,0,1,7};
//printing array using for-each loop for(int i:arr)
System.out.println(i);
}
}
```



```
<terminated> demo
9
0
1
7
```

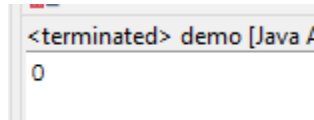
Passing Array to a Method in Java

We can pass the java array to method so that we can reuse the same logic on any array. Program to demonstrate the way of passing an array to method.

```
class demo{
    //creating a method which receives an array as a parameter
    static void min(int arr[]){
        int min=arr[0];
        for(int i=1;i<arr.length;i++){
            if(min>arr[i]) min=arr[i];
        }
        System.out.println(min);
    }

    public static void main(String args[]){
        int a[]={9,0,1,7}; //declaring and initializing an array
        min(a); //passing array to method
    }
}
```

Output:



```
<terminated> demo [Java /
0
```

Anonymous Array in Java

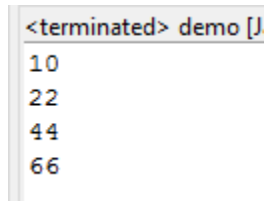
Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

Program to demonstrate the way of passing an anonymous array to method

```
public class demo{
    //creating a method which receives an array as a parameter
    static void printArray(int arr[]){ for(int
        i=0;i<arr.length;i++) System.out.println(arr[i]);
    }

    public static void main(String args[]){
        printArray(new int[]{10,22,44,66}); //passing anonymous array to
method
    }
}
```

Output:



```
<terminated> demo [J
10
22
44
66
```

Returning Array from the Method

We can also return an array from the method in Java.

Program to return an array from the method

```
class demo{
    //creating method which returns an array
    static int[] get(){
        return new int[]{5,10,15,20};
    }

    public static void main(String args[]){
        //calling method which returns an array
        int arr[]=get();
        //printing the values of an array for(int
        i=0;i<arr.length;i++) System.out.println(arr[i]);
    }
}
```

Output:

```
<terminated> demo [
5
10
15
20
```

ArrayIndexOutOfBoundsException

The Java Virtual Machine (JVM) throws an `ArrayIndexOutOfBoundsException` if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

Program to demonstrate the case of `ArrayIndexOutOfBoundsException` in a Java Array

```
public class demo{
    public static void main(String args[]){
        int arr[]={50,60,70,80};
        for(int i=0;i<=arr.length;i++){
            System.out.println(arr[i]);
        }
    }
}
```

Output:

```
<terminated> demo [Cucumber.Automation.demos]
50
60
70
80
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
at Cucumber.Automation.demos.main(demo.java:7)
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare an Multidimensional Array in Java

```
dataType[][] arrayRefVar; (or) dataType
[][]arrayRefVar;          (or)      dataType
arrayRefVar[][];          (or)      dataType
[]arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] twoD_arr = new int[10][20]; //Two dimensional array
```

```
int[][][] threeD_arr = new int[10][20][30]; //Three dimensional array
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Program to illustrate the use of multidimensional 2D array

```
class demo {
    public static void main(String[] args)
    {
        int[][] arr = { { 1, 2 }, { 3, 4 } };

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) { System.out.print(arr[i][j] + "
");
            }
            System.out.println();
        }
    }
}
```

Output

```
<terminated> demo |
1 2
3 4
```

To output all the elements of a Two-Dimensional array, use nested for loops. For this two for loops are

required, One to traverse the rows and another to traverse columns.

Netzwerk Academy

Three – dimensional Array (3D-Array)

Three – dimensional array is a complex form of a multidimensional array. A three – dimensional array can be seen as an array of two – dimensional array for easier understanding.

Indirect Method of Declaration:

Declaration – Syntax:

data_type[][][] array_name = **new** data_type[x][y][z]; For example: **int**[][][]
arr = **new int**[10][20][30];

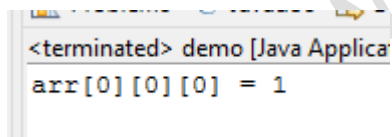
Initialization – Syntax:

array_name[array_index][row_index][column_index] = value; For example:
arr[0][0][0] = 1;

Example:

```
class demo {
    public static void main(String[] args)
    {
        int[][][] arr = new int[10][20][30]; arr[0][0][0] = 1;
        System.out.println("arr[0][0][0] = " + arr[0][0][0]);
    }
}
```

Output:



```
<terminated> demo [Java Applica
arr[0][0][0] = 1
```

Direct Method of Declaration:

Syntax:

```
data_type[][][] array_name = {
    {
        {valueA1R1C1, valueA1R1C2, .....},
        {valueA1R2C1, valueA1R2C2, .....}
    },
    {
        {valueA2R1C1, valueA2R1C2, .....},
        {valueA2R2C1, valueA2R2C2, .....}
    }
};
```

For example: **int**[][][] arr = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };

Example:

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

        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                for (int z = 0; z < 2; z++)
                    System.out.println("arr[" + i
                                       + "]["
                                       + j + "]["
                                       + z + "] = "
                                       + arr[i][j][z]);
    }
}
```

Output:

```
<terminated> demo [Java Ap
arr[0][0][0] = 1
arr[0][0][1] = 2
arr[0][1][0] = 3
arr[0][1][1] = 4
arr[1][0][0] = 5
arr[1][0][1] = 6
arr[1][1][0] = 7
arr[1][1][1] = 8
```

Accessing Elements of Three-Dimensional Arrays

Elements in three-dimensional arrays are commonly referred by $x[i][j][k]$ where 'i' is the array number, 'j' is the row number and 'k' is the column number.

Syntax: `x[array_index][row_index][column_index]` For

example:

```
int[][][] arr = new int[10][20][30]; arr[0][0][0] = 1;
```

The above example represents the element present in the first row and first column of the first array in the declared 3D array.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row_index 2, actual row number is $2+1 = 3$.

Example:

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

        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

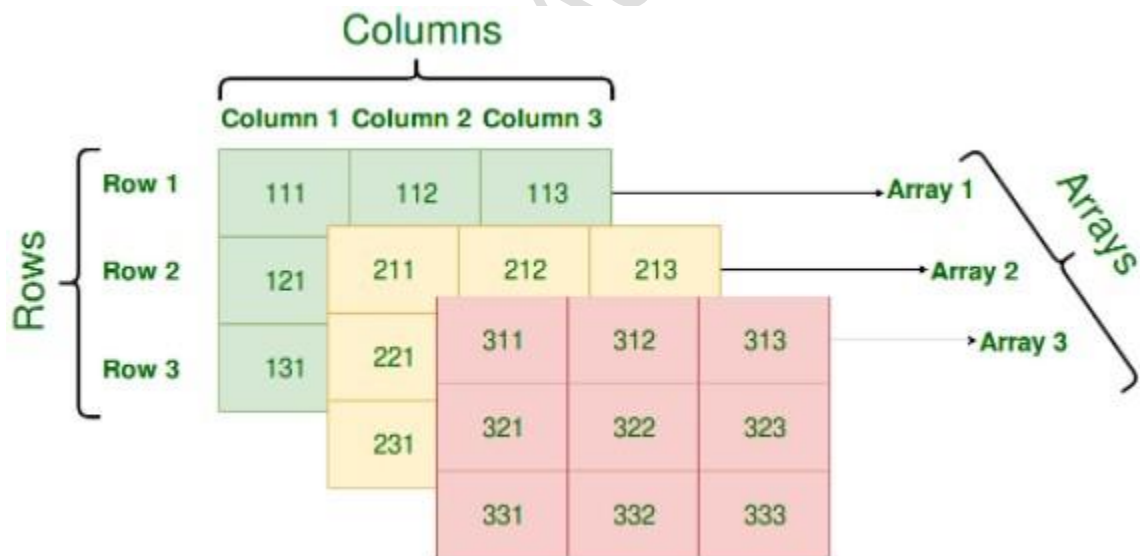
        System.out.println("arr[0][0][0] = " + arr[0][0][0]);
    }
}
```

Output:

```
<terminated> demo [Java Applicat
arr[0][0][0] = 1
```

Representation of 3D array in Tabular Format:

A three – dimensional array can be seen as a tables of arrays with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A three – dimensional array with 3 array containing 3 rows and 3 columns is shown below:



Program to illustrate the use of multidimensional 2D array

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

        int[][][] arr = { { { 1, 2 }, { 3, 4 } },
                           { { 5, 6 }, { 7, 8 } } };
    }
}
```



```
for (int i = 0; i < 2; i++) {
```

Netzwerk Academy

```

    for (int j = 0; j < 2; j++) {

        for (int k = 0; k < 2; k++) {

            System.out.print(arr[i][j][k] + " ");

        }

        System.out.println();

    }
    System.out.println();

}
}

```

Output:

```

<terminated> demo [Java]
1 2
3 4

5 6
7 8

```

To output all the elements of a Three-Dimensional array, use nested for loops. For this three for loops are required, One to traverse the arrays, second to traverse the rows and another to traverse columns.

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

Program to illustrate the jagged array

```

class demo{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][]; arr[0] = new
        int[3];
        arr[1] = new int[4]; arr[2] =
        new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++){
            for(int j=0; j<arr[i].length; j++) arr[i][j] =
                count++;

            //printing the data of a jagged array
            for (int i=0; i<arr.length; i++){
                for (int j=0; j<arr[i].length; j++){
                    System.out.print(arr[i][j]+ " ");
                }
                System.out.println();//new line
            }
        }
    }
}

```

```

<terminated> demo [Java]
0 1 2
3 4 5 6
7 8

```

```
}  
}
```

Netzwerk Academy

Inserting a Multi-dimensional Array during Runtime

This topic is focused on taking user-defined input into a multidimensional array during runtime. It is focused on the user first giving all the input to the program during runtime and after all entered input, the program will give output with respect to each input accordingly. It is useful when the user wishes to make input for multiple Test-Cases with multiple different values first and after all those things done, program will start providing output.

As an example, let's find the total number of even and odd numbers in an input array. Here, we will use the concept of a 2-dimensional array. Here are a few points that explain the use of the various elements in the upcoming code:

- Row integer number is considered as the number of Test-Cases and Column values are considered as values in each Test-Case.
- One for() loop is used for updating Test-Case number and another for() loop is used for taking respective array values.
- As all input entry is done, again two for() loops are used in the same manner to execute the program according to the condition specified.
- The first line of input is the total number of TestCases.
- The second line shows the total number of first array values.
- The third line gives array values and so on.

Program to demonstrate the same

```
import java.util.Scanner;

public class demo {
    public static void main( String[]
        args)
    {
        // Scanner class to take
        // values from console
        Scanner scanner = new Scanner(System.in);

        // totalTestCases = total
        // number of TestCases
        // eachTestCaseValues =
        // values in each TestCase as
        // an Array values
        int totalTestCases, eachTestCaseValues;

        // takes total number of
        // TestCases as integer number totalTestCases =
        scanner.nextInt();

        // An array is formed as row
        // values for total testCases
        int[][] arrayMain = new int[totalTestCases][];
```

```

// for loop to take input of
// values in each TestCase
for (int i = 0; i < arrayMain.length; i++) { eachTestCaseValues =
    scanner.nextInt(); arrayMain[i] = new int[eachTestCaseValues]; for
        (int j = 0; j < arrayMain[i].length; j++) {
            arrayMain[i][j] = scanner.nextInt();
        }
    } // All input entry is done.

// Start executing output
// according to condition provided
for (int i = 0; i < arrayMain.length; i++) {

    // Initialize total number of
    // even & odd numbers to zero
    int nEvenNumbers = 0, nOddNumbers = 0;

    // prints TestCase number with
    // total number of its arguments
    System.out.println(
        "TestCase " + i + " with "
        + arrayMain[i].length + " values:");
    for (int j = 0; j < arrayMain[i].length; j++) {
        System.out.print(arrayMain[i][j] + " ");

        // even & odd counter updated as
        // eligible number is found
        if (arrayMain[i][j] % 2 == 0) {
            nEvenNumbers++;
        }
        else {
            nOddNumbers++;
        }
    }
    System.out.println();

    // Prints total numbers of
    // even & odd System.out.println(
        "Total Even numbers: " + nEvenNumbers
        + ", Total Odd numbers: " + nOddNumbers);
    }
}

```

Output:

```

<terminated> demo [Java Application] C:\Program Files\Java\jdk-14
2
2
1 2
3
1 2 3
TestCase 0 with 2 values:
1 2
Total Even numbers: 1, Total Odd numbers: 1
TestCase 1 with 3 values:
1 2 3
Total Even numbers: 1, Total Odd numbers: 2

```

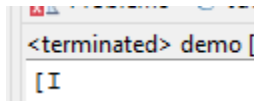
What is the class name of Java array?

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by `getClass().getName()` method on the object.

Program to get the class name of array in Java

```
class demo{
    public static void main(String args[]){
        //declaration and initialization of array
        int arr[]={4,4,5};
        //getting the class name of Java array Class
        c=arr.getClass();
        String name=c.getName();
        //printing the class name of Java array
        System.out.println(name);
    }
}
```

Output:



```
<terminated> demo [
[I
```

Copying a Java Array

We can copy an array to another by the `arraycopy()` method of `System` class.

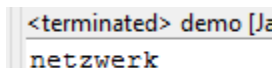
Syntax of `arraycopy` method

```
public static void arraycopy(
    Object src, int srcPos, Object dest, int destPos, int length
)
```

Program to copy a source array into a destination array in Java

```
class demo {
    public static void main(String[] args) {
        //declaring a source array
        char[] copyFrom = { 'r', 'q', 'o', 'n', 'e', 't', 'z',
                           'w', 'e', 'r', 'k', 'i', 't' };
        //declaring a destination array
        char[] copyTo = new char[8]; //max char to copy
        //copying array using System.arraycopy() method System.arraycopy(copyFrom, 3, copyTo,
        0, 8); //Copy starts from
        3rd char till 3rd Char + 8
        //printing the destination array System.out.println(String.valueOf(copyTo));
    }
}
```

Output:



```
<terminated> demo [Ja
netzwerk
```

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

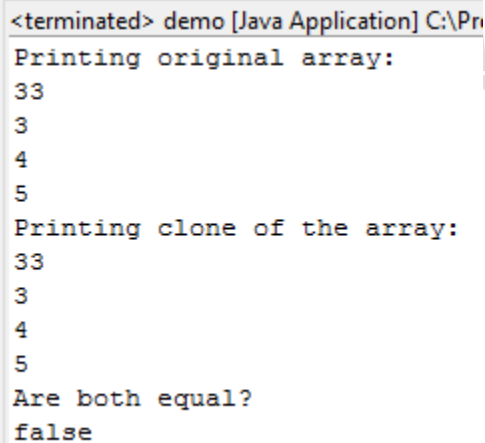
Program to clone the array

```
class demo{
public static void main(String args[]){
int arr[]={33,3,4,5}; System.out.println("Printing original array:");
for(int i:arr)
System.out.println(i);

System.out.println("Printing clone of the array:");
int carr[]=arr.clone(); for(int
i:carr) System.out.println(i);

System.out.println("Are both equal?");
System.out.println(arr==carr);
}
}
```

Output:



```
<terminated> demo [Java Application] C:\Pr
Printing original array:
33
3
4
5
Printing clone of the array:
33
3
4
5
Are both equal?
false
```


Addition of 2 Matrices in Java

Program to demonstrate the addition of two matrices in Java

```
class demo{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){ for(int
j=0;j<3;j++){ c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}
}
```

Output:

```
<terminated> demo [
2 6 8
6 8 10
```

Multiplication of 2 Matrices in Java

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given.

$$\text{Matrix 1} \begin{Bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{Bmatrix} \quad \text{Matrix 2} \begin{Bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{Bmatrix}$$

$$\begin{matrix} \text{Matrix 1} \\ * \\ \text{Matrix 2} \end{matrix} \begin{Bmatrix} 1*1+1*2+1*3 & 1*1+1*2+1*3 & 1*1+1*2+1*3 \\ 2*1+2*2+2*3 & 2*1+2*2+2*3 & 2*1+2*2+2*3 \\ 3*1+3*2+3*3 & 3*1+3*2+3*3 & 3*1+3*2+3*3 \end{Bmatrix}$$

$$\begin{matrix} \text{Matrix 1} \\ * \\ \text{Matrix 2} \end{matrix} \begin{Bmatrix} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{Bmatrix}$$

Program to multiply two matrices

```
public class demo{
    public static void main(String args[]){
        //creating two matrices
        int a[][]={{1,1,1},{2,2,2},{3,3,3}};
        int b[][]={{1,1,1},{2,2,2},{3,3,3}};

        //creating another matrix to store the multiplication of two matrices
        int c[][]=new int[3][3];           //3 rows and 3 columns

        //multiplying and printing multiplication of 2 matrices
        for(int i=0;i<3;i++){ for(int
        j=0;j<3;j++){ c[i][j]=0;
        for(int k=0;k<3;k++)
        {
            c[i][j]+=a[i][k]*b[k][j];
        } //end of k loop
        System.out.print(c[i][j]+" ");           //printing matrix element
        } //end of j loop System.out.println();//new
        line
    }
    }
}
```

Output:

```
<terminated> demo [Ja
6 6 6
12 12 12
18 18 18
```