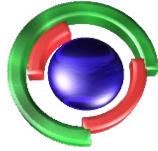
FUTURE VISION BIE

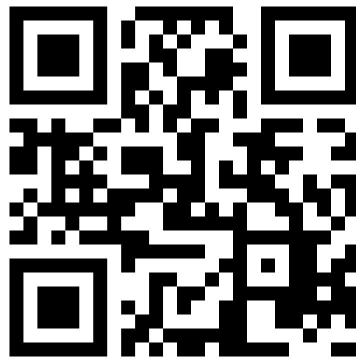
One Stop for All Study Materials & Lab Programs



Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -2 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

https://hemanthrajhemu.github.io

MODULE 2 CLASS AND OBJECTS (CONTD)

The following concepts should be learn in this Module

Objects and arrays, Namespaces, Nested classes, Constructors, Destructors. **Introduction to Java**: Java's magic: the Byte code; Java Development Kit (JDK); the Java Buzzwords, Object-oriented programming; Simple Java programs. Data types, variables and arrays, Operators, Control Statements.

Text book 1:Ch 2: 2.4 to 2.6Ch 4: 4.1 to 4.2

Text book 2: Ch:1 Ch: 2 Ch:3 Ch:4 Ch:5 , RBT: L1, L2

OBJECTS AND ARRAYS:

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

The syntax for declaring an array of objects is

class_name array_name [size] ;

```
class Employee
{
char name[30];
int age;
public:
void getdata(void);
void putdata(void);
};
void Employee:: getdata(void)
{
cout<<"Enter Name and Age:";
cin>>name>>age;
}
void Employee:: putdata(void)
{
cout<<name<<"\t"<<age<<endl;
}
void main()
```


https://hemanthrajhemu.github.io

```
{
Employee e[5];
int i;
for(i=0; i<5; i++)
{
e[i].getdata();
}
for(i=0; i<5; i++)
{
e[i].putdata();
}
}
```

NAMESPACE:

Namespaces – enable C++ programmer to prevent pollution of global namespace that lead to name clashes.

Global namespace refer to the entire source code. It includes all the directly and indirectly included header files. By default, name of each class is visible in the source code i.e. in the global space. This can lead to problems.

Namespace is used to define a scope where identifiers like variables, functions, classes, etc are declared. The main purpose of using a namespace is to prevent ambiguity that may occur when two identifiers have same name.

Consider

```
A1.h  A2.h

class A

{

//body of A

};      class A

{

//body of A

};
```

Main_prog.cpp

https://hemanthrajhemu.github.io

```
#include "A1.h" #include "A2.h" main()
{
A Aobj;       //ERROR: Ambiguity error due to multiple definitions of A
}
```

Syntax of Namespace Definition:

The member can be accessed in the program as,

using namespace namepace_name namespace_name::member_name;

```
/*A1.h*/
namespace A1
{
class A
{
};
}       /*end of namespace A1.h*/      /*A2.h*/
namespace A2
{
class A
{
};
}       /*end of namespace A2.h*/
```

The using directive enable us to make class definition inside A namespace visible so that qualifying the name of referred Class by name of namespace is no longer required. Code below tells how this is done.

```
#include "A1.h" #include "A2.h" void main()
```


https://hemanthrajhemu.github.io

```
{
using namespace A1;
A1::A Aobj1;        //ok: Aobj1 is an object of class defined in A1.h
A2::A Aobj2;        //ok: Aobj2 is an object of class defined in A1.h
}
```

Rules for namespace:

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don‟t have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.

using namespace std;

The using namespace statement specifies that the members defined in std namespace will be used frequently throughout the program.

CONSTRUCTORS:

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. Constructors can be very useful for setting initial values for certain member variables.

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created.

Characteristics of a constructor

- They should be declared in the public section.
- They are invoked directly when an object is created.
- They don‟t have return type, not even void and hence can‟t return any values.
- They can‟t be inherited; through a derived class, can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors can‟t be virtual.
- Constructors can be inside the class definition or outside the class definition.
- Constructor can‟t be friend function.

https://hemanthrajhemu.github.io

- They make implicit calls to the operators new and delete when memory allocation is required.
- When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor.

Different Types of Constructor

Default Constructor:-

Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of Return Type.

```
class Add
{
int x, y, z;
public:
Add(); // Default Constructor
void display(void);
};
Add::Add()
{
x=6;
y=5;
}
void Add :: display()
{
cout<< x+y;
}
void main()
{
Add a;
a.display();
}
```

Output:
11

https://hemanthrajhemu.github.io

Parameterized Constructor:-

This is Another type Constructor which has some Arguments and same name as class name .We have to create object of Class by passing some Arguments at the time of creating object with the name of class.

```
 class Add
{
Int x, y, z;
public:
Add(int, int);
void display(void);
};
Add :: Add(int a, int b)
{
x=a;
y=b;
}
void Add :: display()
{
cout<<x+y;
}
void main()
{
Add a(5, 6);
a.display();
}
```

Output:

11

A parameterized constructor can be called:

(i) Implicitly: Add a(5, 6);

(ii) Explicitly : Add a=Add(5, 6);

Copy Constructor:-

This is also Another type of Constructor. In this Constructor we pass the object of class into the Another Object of Same Class. As name Suggests you Copy, means Copy the values of one Object into the another Object of Class .

When we are using or passing an Object to a Constructor then we must have to use the & Ampersand or Address Operator.

https://hemanthrajhemu.github.io

```
class Add
{
int x, y, z;
public:
Add(int a, int b)
{
x=a;
y=b;
}


Add(Add &);
void display(void);
};
Add :: Add(Add &p)
{
x=p.x;
y=p.y;
cout<<"Value of x and y for new object: "<<x<<" and
"<<y<<endl;
}
void Add :: display()
{
cout<<x+y;
}
void main()
{
Add a(5, 6);
Add b(a);
b.display();
}
```

Output:

Value of x and y for new object are 5 and 6 11

DESTRUCTORS

- It is a special member function which is executed automatically when an object is destroyed.
- Its name is same as class name but it should be preceded by the symbol ~.
- It cannot be overloaded as it takes no argument.
- It is used to delete the memory space occupied by an object.
- It has no return type.
- It should be declared in the public section of the class.

https://hemanthrajhemu.github.io

```
class A
{
A()
{
cout << "Constructor called";
}
~A()
{
cout << "Destructor called";
}

};
int main()
{
A obj1; // Constructor Called
int x=1
if(x)
{
A obj2; // Constructor Called
} // Destructor Called for obj2
} // Destructor called for obj1
```

NESTED CLASSES:

A class can be defined inside another class. Such a class is known as Nested Class. The class that contains the nested class is known as enclosing class. Nested classes can be defined in the private, protected, or public sections of the enclosing class.

- A nested class is declared inside another class.
- The scope of inner class is restricted by the outer class.
- While declaring an object of inner class, the name of the inner class must be preceded by the outer class name and scope resolution operator.

A nested class is created if it does not have any relevance outside its enclosing class. By defining class as a nested class, name collision can be avoided. That is, if there is class B defined as global class, its name will not clash with the nested class B. The size of the object of an enclosing class is not affected by the presence of nested classes.

https://hemanthrajhemu.github.io

How are the member functions of a nested class defined?

Member functions of a nested class can be defined outside the definition of enclosing class.

Syntax:

```
class A
{
public:
class B
{
public:
void BTest();
};
};      // Defining function outside the class



void A :: B :: BTest()
{
//function body
}
```

```
#include <iostream.h>
 class Nest
{       public:
class Display
{       private:
int s;
public:
```


https://hemanthrajhemu.github.io

```
void sum( int a, int b)
{       s =a+b;
}
void show( )
{       cout << "\nSum of a and b is:: " << s;
}
};
};
void main()
{       Nest::Display x;
x.sum(12, 10);
x.show();
}
```

Sum of a and b is::22

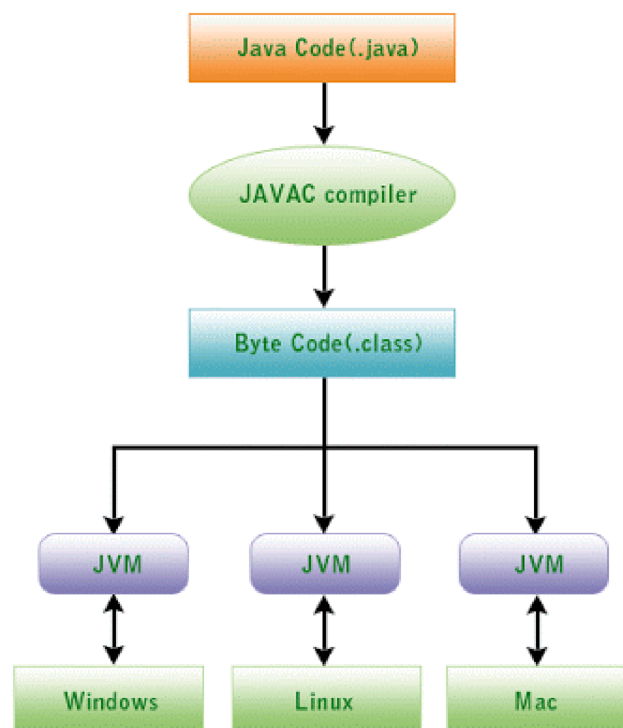
https://hemanthrajhemu.github.io

INTRODUCTION TO JAVA

Java Byte Code:

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM)*.In essence, the original JVM was designed as an interpreter for bytecode.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.



Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs. Programming code, once compiled, is run through a virtual machine instead of the computer's processor. By using this approach, source code can be run on any platform once it has been compiled and run through the virtual machine.

Bytecode is the compiled format for Java programs. Once a Java program has been converted to bytecode, it can be transferred across a network and executed

https://hemanthrajhemu.github.io

by Java Virtual Machine (JVM). Bytecode files generally have a .class extension.

A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. Rather than being interpreted one instruction at a time, Java bytecode can be recompiled at each particular system platform by a just-in-time compiler. Usually, this will enable the Java program to run faster.

Java Development Kit (JDK):

Java Development Kit contains two parts. One part contains the utilities like **javac**, debugger, **jar** which helps in compiling the source code (**.java** files) into byte code (**.class** files) and debug the programs. The other part is the **JRE**, which contains the utilities like **java** which help in running/executing the byte code. If we want to write programs and run them, then we need the JDK installed.

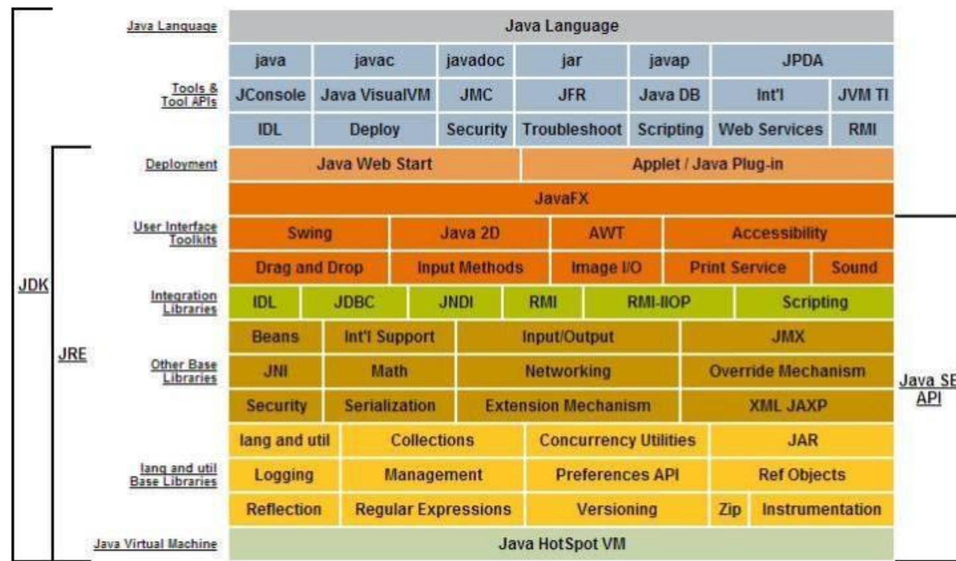
Java Run-time Environment (JRE):

Java Run-time Environment helps in running the programs. **JRE** contains the **JVM**, the java classes/packages and the run-time libraries. If we do not want to write programs, but only execute the programs written by others, then **JRE** alone will be sufficient.

Java Virtual Machine (JVM):

Java Virtual Machine is important part of the **JRE**, which actually runs the programs (**.class** files), it uses the java class libraries and the run-time libraries to execute those programs. Every operating system(OS) or **platform** will have a different **JVM**.

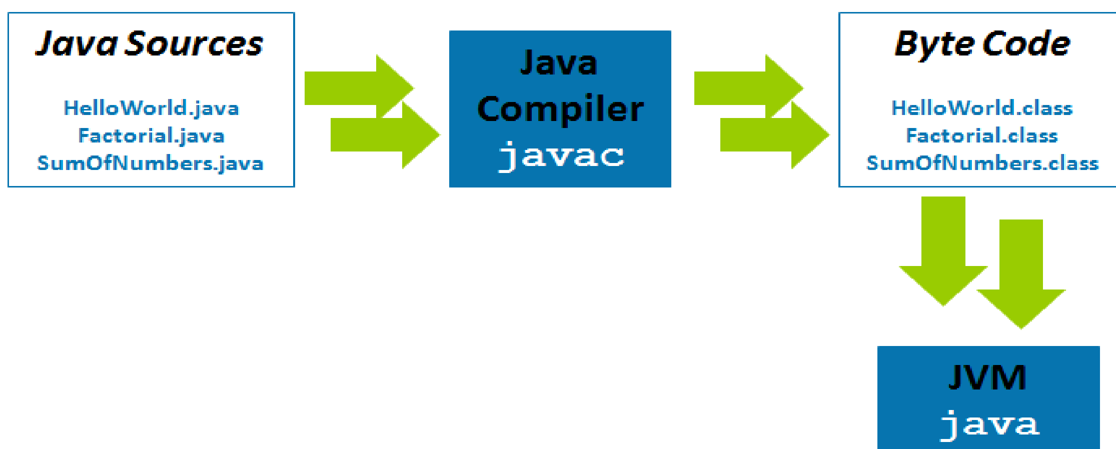
https://hemanthrajhemu.github.io



Just In Time Compiler (JIT):

JIT is a module inside the **JVM** which helps in **compiling** certain parts of byte code into the **machine code** for higher performance. Note that only certain parts of byte code will be compiled to the machine code, the other parts are usually **interpreted** and executed.

Java is distributed in two packages - **JDK** and **JRE**. When **JDK** is installed it also contains the **JRE**, **JVM** and **JIT** apart from the compiler, debugging tools. When **JRE** is installed it contains the **JVM** and **JIT** and the class libraries. **javac** helps in compiling the program and **java** helps in running the program. When the words **Java Compiler**, **Compiler** or **javac** is used it refers to **javac**, when the words **JRE**, **Run-time Enviroment**, **JVM**, **Virtual Machine** are used, it refers to **java**.



Write (Compile) Once and Run Anywhere (WORA)

https://hemanthrajhemu.github.io

This terminology was given by Sun Microsystem for their programming language - Java. According to this concept, the same code must run on any machine and hence the source code needs to be portable. So Java allows run Java bytecode on any machine irrespective of the machine or the hardware, using JVM (Java Virtual Machine). The bytecode generated by the compiler is not platform-specific and hence takes help of JVM to run on a wide range of machines. So we can call Java programs as a write once and run on any machine residing anywhere.

Java Buzz Words

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. If you already understand the basic concepts of object-oriented programming like C++, learning Java will be even easier.

Object Oriented:

In Java, everything is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Platform Independent:

Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

https://hemanthrajhemu.github.io

Secure: With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architecture-neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable: Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and de-allocation for you. Java helps in this area by providing object-oriented exception handling.

Multithreaded: With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

High Performance: With the use of Just-In-Time compilers, Java enables high performance.
Distributed: Java is designed for the distributed environment of the internet. Java also supports Remote Method Invocation (RMI). It handles TCP/IP protocols.

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

https://hemanthrajhemu.github.io

History of Java

James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called 'Oak' after an oak tree that stood outside Gosling's office, also went by the name 'Green' and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.
On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Object Oriented Paradigm

Abstraction

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction.
From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.
For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Three OO Concepts:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. It acts as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

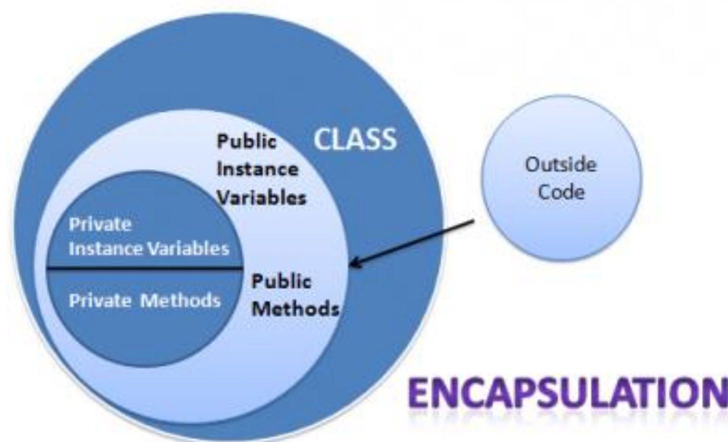
https://hemanthrajhemu.github.io

A class defines the structure and behaviour (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class. For this reason, objects are sometimes referred to as instances of a class.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as **member variables or instance variables**. The code that operates on that data is referred to as **member methods or just methods.**

Each method or variable in a class may be marked private or public.

- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class.

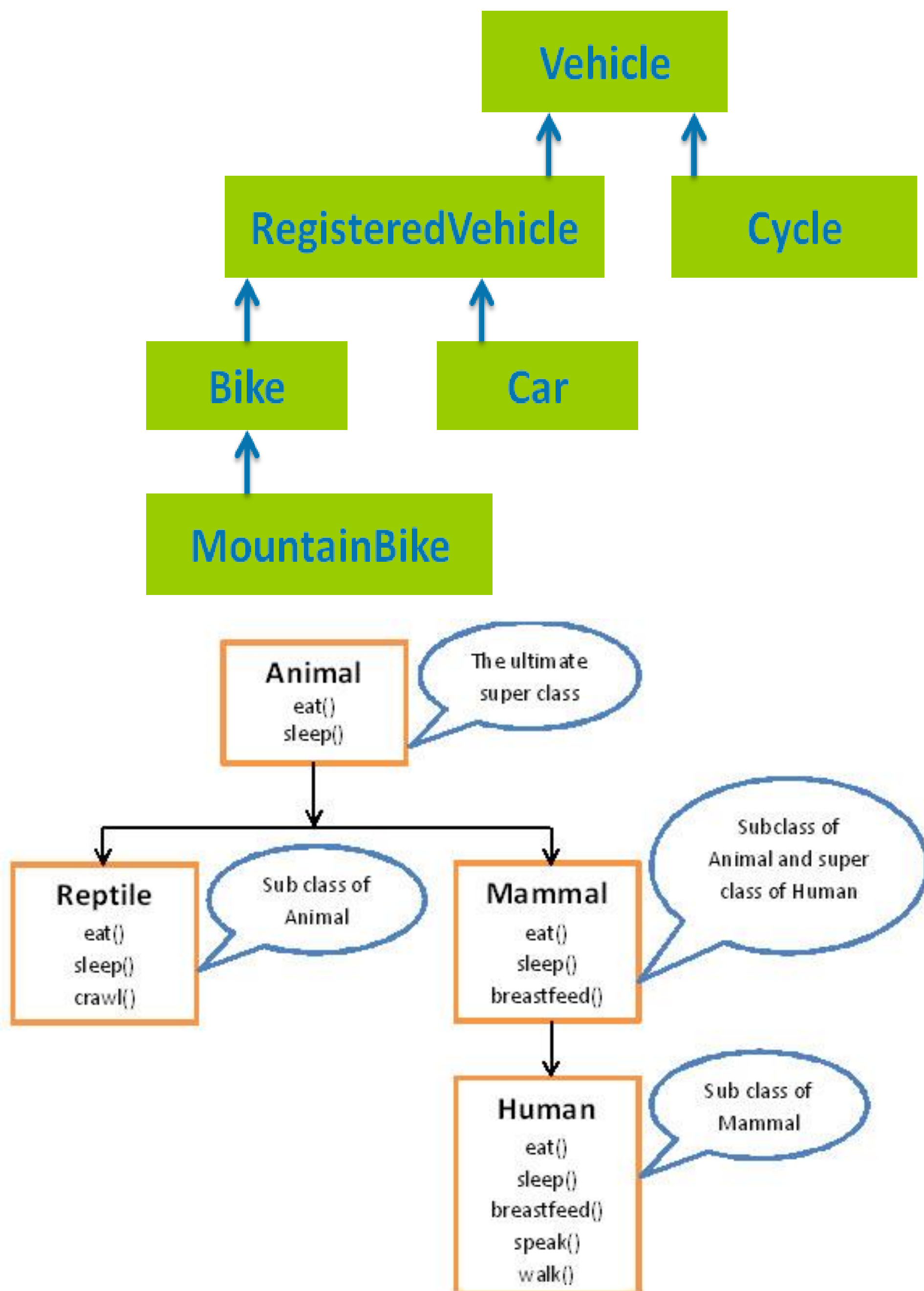


Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

The class which inherits the properties of other is known as **subclass** (derived class, child class) and the class whose properties are inherited is known as **superclass** (base class, parent class).

https://hemanthrajhemu.github.io



Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. **Polymorphism in java** is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word

https://hemanthrajhemu.github.io

"poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java:

- compile time polymorphism and
- Runtime polymorphism.

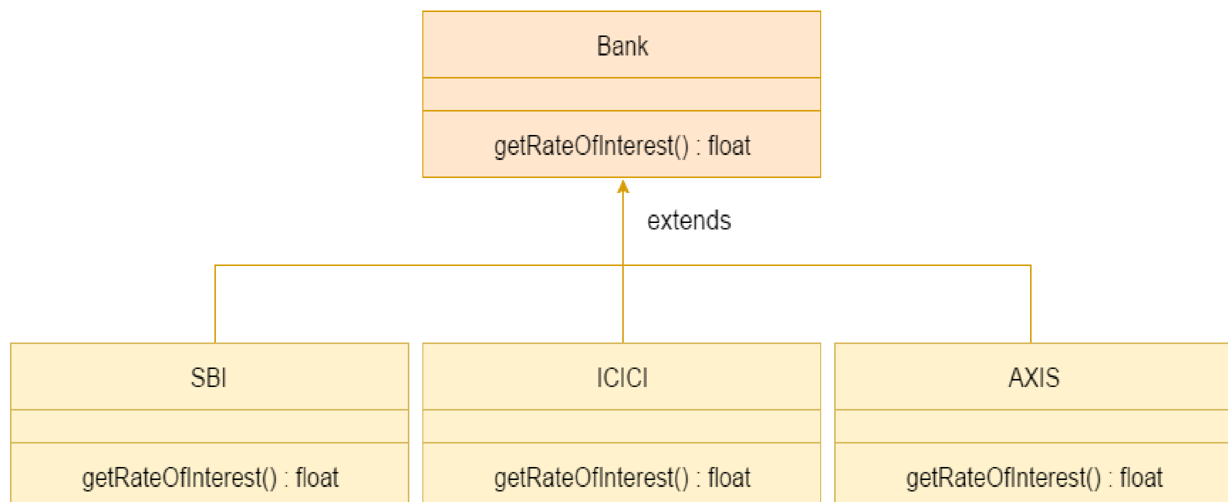
We can perform polymorphism in java by **method overloading and method overriding.**

Example: Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

```
public class Dog extends Animal {

    public void makeSound() {
        System.out.println("Woof!");
    }

    public void makeSound(boolean injured) {
        System.out.println("Whimper");
    }
}
```



Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scalable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you

https://hemanthrajhemu.github.io

have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Java Basics

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object -** Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviour such as wagging their tail, barking, eating. An object is an instance of a class.

- **Class -** A class can be defined as a template/blueprint that describes the behaviour/state that the object of its type supports.

- **Methods -** A method is basically behaviour. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

- **Instance Variables -** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

```
public class MyFirstJavaProgram {
/* This is my first java program.
This will print 'Hello World' as the output */


  public static void main(String []args) {
     System.out.println("Hello World"); // prints Hello World
}
}
```

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
    Hello World
```


https://hemanthrajhemu.github.io

Java supports three styles of comments.

1) The one shown at the top of the program is called a multiline comment. This type of comment must begin with /* and end with */.
2) Single line comment . //
3) Documentation Comment

public static void main(String args[]) {

All Java applications begin execution by calling main( ).
When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.
The keyword static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java Virtual Machine before any objects are made.

System.out.println("This is a simple Java program.");

This line outputs the string "This is a simple Java program." followed by a new line on the
screen. In this case, println( )displays the string which is passed to it.
System is a predefined class that provides access to the system, and out is the output stream that is connected to the console.

DATATYPES

There are eight primitive datatypes supported by Java.

Data Type	Range	Default size
Boolean	False/ true	1 bit
char	0-65536	2 byte
byte	–128 to 127	1 byte
short	–32,768 to 32,767	2 byte
int	–2,147,483,648 to 2,147,483,647	4 byte
long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 byte
float	1.4e–045 to 3.4e+038	4 byte
double	4.9e–324 to 1.8e+308	8 byte

Integers: Java does not support unsigned, positive-only integers.

https://hemanthrajhemu.github.io

Char: Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.

```
public class JavaCharExample {

    public static void main(String[] args) {
        char ch1 = 'a';
        char ch2 = 65; /* ASCII code of 'A'*/

        System.out.println("Value of char variable ch1 is :" +
ch1);
        System.out.println("Value of char variable ch2 is :" +
ch2);
    }
}
```

```
Output would be
Value of char variable ch1 is :a
Value of char variable ch2 is :A
```

- **Byte** are useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

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

byte b1 = 100;
byte b2 = 20;

System.out.println("Value of byte variable b1 is :" + b1);
System.out.println("Value of byte variable b1 is :" + b2);
}
}
```

```
Output would be
Value of byte variable b1 is :100
Value of byte variable b1 is :20
```

- **Boolean takes either True or false**

https://hemanthrajhemu.github.io

```
public class JavaBooleanExample {

public static void main(String[] args) {

    boolean b1 = true;
    boolean b2 = false;
    boolean b3 = (10 > 2)? true:false;

System.out.println("Value of boolean variable b1 is :" + b1);
System.out.println("Value of boolean variable b2 is :" + b2);
System.out.println("Value of boolean variable b3 is :" + b3);
}
}
```

```
Output would be
Value of boolean variable b1 is :true
Value of boolean variable b2 is :false
Value of boolean variable b3 is :true
```

- **Floating**-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.

```
import java.util.*;

public class JavaFloatExample {

        public static void main(String[] args) {
                float f = 10.4f;
                System.out.println("Value of float variable f is :" + f);
        }
}
```

```
Output would be
 Value of float variable f is :10.4
```

- **Double** precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values.

```
        public class JavaDoubleExample {

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


https://hemanthrajhemu.github.io

``
double d = 1232.44;
System.out.println("Value of double variable d is :" + d);
}
}
``

Output would be Value of double variable f is :1232.44

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable.

For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

https://hemanthrajhemu.github.io

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Form feed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

VARIABLES

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory;

Declaring a Variable

data type variable [= value][, variable [= value] ...] ;

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;                    // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;                          // initializes a byte type variable B.
double pi = 3.14159;       // declares and assigns a value of PI.
char a = 'a';  // the char variable a is initialized with value 'a'
```

There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/Static variables

Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are implemented at stack level internally.

https://hemanthrajhemu.github.io

- Access modifiers cannot be used for local variables.

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Access modifiers can be given for instance variables.

Class/static Variables

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
Ex:

```
class dyn {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse

The Scope and Lifetime of Variables

Java allows variables to be declared within any block. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

https://hemanthrajhemu.github.io

In Java, the two major scopes are those defined by a class and those defined by a method.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

To understand the effect of nested scopes, consider the following program:

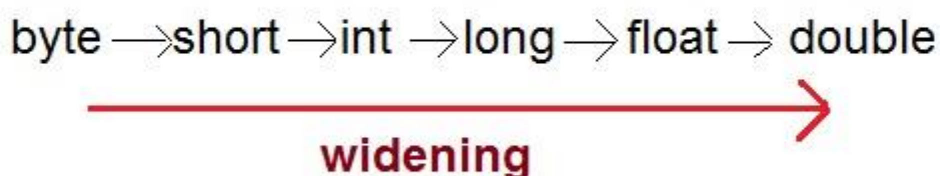
```
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10;
if(x == 10) { // start new scope
int y = 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}
}
```

Output

```
x and y: 10 20
x is 22
```

Type Conversion and Casting

- **Widening Casting(Implicit)**



https://hemanthrajhemu.github.io

- **Narrowing Casting(Explicitly done)**

1) **Java's Automatic Conversions (Implicit Conversion)**

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

The two types are compatible. The destination type is larger than the source type. Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Ex:

```
//64 bit long integer

long l;

//32 bit long integer

int i;

l=i;
```

2) **Casting Incompatible Types**

Although the automatic type conversions are helpful, they will not fulfill all needs. To create a ***narrowing*** conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

It has this general form: **(target-type) value**

```
public class Test
{
   public static void main(String[] args)
   {
     double d = 100.04;
```


https://hemanthrajhemu.github.io

```
    long l = (long)d;  //explicit type casting required
    int i = (int)l;    //explicit type casting required

    System.out.println("Double value "+d);
    System.out.println("Long value "+l);
    System.out.println("Int value "+i);

  }

}
```

```
Output :
Double value 100.04
Long value 100
Int value 100
```

Automatic Type Promotion in Expressions

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term a * b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the sub expression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!

byte b = 50;
b = (byte)(b * 2); //which yields the correct value of 100.

```
class Promote {
public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
```


https://hemanthrajhemu.github.io

```
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);

}
}
Output: will be of double data type
```

Type Promotion Rules

1.All byte, short and char values are promoted to int.
2.If one operand is a long, the whole expression is promoted to long.
3.If one operand is a float, the entire expression is promoted to float.
4.If any of the operands is double, the result is double.

ARRAYS

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.

One-Dimensional Arrays

A one-dimensional array is essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional
array declaration is:

```
datatype identifier [ ];
```

Or

```
datatype[ ] identifier;
```

Ex: int month_days[];

It declares an array variable but do not allocate any memory. New is a special operator that allocates memory.

array-var = new type[size]; // (new will automatically be initialized to zero)

OR

dataType[] arrayVar = new dataType[arraySize];

https://hemanthrajhemu.github.io

```
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

Arrays can be initialized when they are declared. There is no need to use new.

```
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}

Output:
April has 30 days
```

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays.

int twoD[][] = new int[4][5];

Ex:

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
```


https://hemanthrajhemu.github.io

```
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Ex:

```
String[][] sampleData = {   {"a", "b", "c", "d"}, {"e", "f", "g", "h"}, {"i", "j",
"k", "l"},
       {"m", "n", "o", "p"} };
```

OPERATORS IN JAVA

Java provides a rich operator environment. Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

Arithmetic Operators
Relational Operators
Bitwise Operators
Logical Operators
Ternary Operator and
Assignment Operator.

https://hemanthrajhemu.github.io

<table>
<tr><th>Operators</th><th>Precedence</th></tr>
<tr><td>Postfix</td><td>expr++ expr--</td></tr>
<tr><td>Unary</td><td>++expr --expr +expr -expr ~ !</td></tr>
<tr><td>Multiplicative</td><td>* / %</td></tr>
<tr><td>Additive</td><td>+ -</td></tr>
<tr><td>Shift</td><td><< >> >>></td></tr>
<tr><td>Relational</td><td>< > <= >= instance of</td></tr>
<tr><td>Equality</td><td>== !=</td></tr>
<tr><td>bitwise AND</td><td>&</td></tr>
<tr><td>bitwise exclusive OR</td><td>^</td></tr>
<tr><td>bitwise inclusive OR</td><td>|</td></tr>
<tr><td>logical AND</td><td>&&</td></tr>
<tr><td>logical OR</td><td>||</td></tr>
<tr><td>Ternary</td><td>? :</td></tr>
<tr><td>Assignment</td><td>= += -= *= /= %= &= ^= |= <<= >>= >>>=</td></tr>
</table>

Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types.

- The unary minus operator negates its single operand.
- The unary plus operator simply returns the value of its operand.

```
class OperatorExample{
public static void main(String args[]){    int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

```
Output
15
5
50
2
0
```


https://hemanthrajhemu.github.io

The Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the %:

```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

```
x mod 10 = 2
y mod 10 = 2.25
```

Note: **RESULT OF ARITHMETIC EXPRESSIONIS ALWAYS**

MAX(int, type of a and type of b)

Ex: byte a=10;
byte b= 20;
c= a+b; // the result will be integer type

Arithmetic Compound Assignment Operators [Shorthand assignment]

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

***var op= expression*;**

```
class OperatorExample{
   public static void main(String args[]){
   int a=10;
   int b=20;
   a+=4;//a=a+4 (a=10+4)
   b-=4;//b=b-4 (b=20-4)
   System.out.println(a);
   System.out.println(b);
   }}
```

The Relational Operators

https://hemanthrajhemu.github.io

The *relational operators* determine the relationship that one operand has to the other. The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

operator	Description
==	Check if two operands are equal
!=	Check if two operands are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands and relational expressions. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value. The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer.

<table>
<tr><th>Operator</th><th>Result</th></tr>
<tr><td>&</td><td>Logical AND</td></tr>
<tr><td>|</td><td>Logical OR</td></tr>
<tr><td>^</td><td>Logical XOR (exclusive OR)</td></tr>
<tr><td>||</td><td>Short-circuit OR</td></tr>
<tr><td>&&</td><td>Short-circuit AND</td></tr>
<tr><td>!</td><td>Logical unary NOT</td></tr>
<tr><td>&=</td><td>AND assignment</td></tr>
<tr><td>|=</td><td>OR assignment</td></tr>
<tr><td>^=</td><td>XOR assignment</td></tr>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>?:</td><td>Ternary if-then-else</td></tr>
</table>

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. The OR operator results in true when A is true, no matter

https://hemanthrajhemu.github.io

what B is. Similarly, the AND operator results in false when A is false, no matter what B is.
Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if ( denom != 0 && num / denom >10)
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code was written using the single “&” version of AND, both sides would have to be evaluated, causing a run-time exception when denom is zero.

```
class ShortCircuitAnd
{
   public static void main(String arg[])
   {
      int c = 0, d = 100, e = 50; // LINE A
      if( c == 1 && e++ < 100 )
      {
         d = 150;
      }
      System.out.println("e = " + e );
      }
}
```

```
OUTPUT
e = 50
```


https://hemanthrajhemu.github.io

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

<table>
<tr><th>operator</th><th>description</th></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise exclusive OR</td></tr>
<tr><td><<</td><td>left shift</td></tr>
<tr><td>>></td><td>right shift</td></tr>
<tr><td>>>></td><td>Right Shift zero fill</td></tr>
<tr><td>&=</td><td>Bitwise AND assignment</td></tr>
<tr><td>|=</td><td>Bitwise OR assignment</td></tr>
<tr><td>^=</td><td>Bitwise X-OR assignment</td></tr>
<tr><td>>>>=</td><td>Shift right zero fill assignment</td></tr>
<tr><td>>>=</td><td>Shift right assignment</td></tr>
<tr><td><<=</td><td>Shift left assignment</td></tr>
</table>

<table>
<tr><th>A</th><th>B</th><th>~A</th><th>A & B</th><th>A | B</th><th>A ^ B</th></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Java AND Operator: Logical && vs Bitwise &

```
class OperatorExample{
   public static void main(String args[]){
   int a=10;
   int b=5;
   int c=20;
```


https://hemanthrajhemu.github.io

```
   System.out.println(a<b && a++<c);        //false && true = false
   System.out.println(a);                     //10 because second condition
is not checked
   System.out.println(a<b & a++<c);         //false && true = false
   System.out.println(a);                //11 because second condition is
checked
   }}
```

Output:

```
false
10
false
11
```

Java OR Operator: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
   public static void main(String args[]){
   int a=10;
   int b=5;
   int c=20;
   System.out.println(a>b||a<c);          //true || true = true

   System.out.println(a>b|a<c);           //true | true = true

   System.out.println(a>b||a++<c);        //true || true = true
   System.out.println(a);                 //10 because second condition is not
checked
   System.out.println(a>b|a++<c);         //true | true = true
   System.out.println(a);                    //11 because second condition is
checked
   }}
```


https://hemanthrajhemu.github.io

Output:

```
true
true
true
10
true
11
```

Left Shift & Right Shift

```
   class OperatorExample
{
   public static void main(String args[]){
   System.out.println(10<<2);      //10*2^2=10*4=40
   System.out.println(10<<3);      //10*2^3=10*8=80
   System.out.println(10>>2);      //10/2^2=10/4=2
   System.out.println(20>>2);      //20/2^2=20/4=5
   System.out.println(20>>3);      //20/2^3=20/8=2
   }
}
```

Java Shift Operator Example: >> vs >>>

>>> is also known as Unsigned Right Shift. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift*.

int x = 13 >>> 1;
Out put : 6

// 13 = 00000000000000000000000000001101
// 6 = 00000000000000000000000000000110

y = -8 >>> 2;
Output : 1073741822 // -8 = 11111111111111111111111111111000

https://hemanthrajhemu.github.io

// 1073741822 =
00111111111111111111111111111110

The following code fragment demonstrates the >>>. Here, a is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int

```
class OperatorExample{
public static void main(String args[]){
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For nagative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
  }
}
```

Output

```
5
5
-5
1073741819
```

Java Ternary Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered.

The ? has this general form:

https://hemanthrajhemu.github.io

expression1 ? expression2 : expression3

Here, expression1 can be any expression that Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated.

```
class OperatorExample{
  public static void main(String args[]){
  int a=2;
  int b=5;
  int min=(a<b)?a:b;
  System.out.println(min);
  }}
```

Output:

2

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.

```
class IncDec {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

Output

a = 2
b = 3
c = 4
d = 1

CONTROL STATEMENTS

https://hemanthrajhemu.github.io

If- else:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (condition)
statement1;
else
statement2;

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

```
public class IfExample {
public static void main(String[] args) {
   int age=20;
   if(age>18)
     {
      System.out.print("Eligible to vote");
      }
}
}
```

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

Syntax :

```
if (condition)
{
        if (condition){
                //Do something
        }
//Do something
}
```

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c;          // associated with this else
}
else a = d;
```


https://hemanthrajhemu.github.io

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif
ladder. It looks like this:

Syntax: if(condition)

```
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

```
public class ControlFlowDemo
{
   public static void main(String[] args)
   {
      char ch = 'o';

      if (ch == 'a' || ch == 'A')
         System.out.println(ch + " is vowel.");
      else if (ch == 'e' || ch == 'E')
         System.out.println(ch + " is vowel.");
      else if (ch == 'i' || ch == 'I')
         System.out.println(ch + " is vowel.");
      else if (ch == 'o' || ch == 'O')
         System.out.println(ch + " is vowel.");
      else if (ch == 'u' || ch == 'U')
         System.out.println(ch + " is vowel.");
      else
         System.out.println(ch + " is a consonant.");
          }
}
```

Switch

The **switch** statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

https://hemanthrajhemu.github.io

Syntax: switch (*expression*)

```
{
        case value1:
        // statement sequence
        break;
        case value2:
        // statement sequence
        break;
        .
        .
        .
        case valueN :
        // statement sequence
        break;
        default:
        // default statement sequence
}
```

expression must be of type byte, short, int, char, or enumerated data type(String).

```
class StringSwitch {
public static void main(String args[]) {
String str = "two";
 switch(str)
{
 case "one":
                System.out.println("one");
                break;
case "two":
                System.out.println("two");
                break;
case "three":
                 System.out.println("three");
                 break;
default:
               System.out.println("no match");
               break;
}}}
```

Output : two

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


https://hemanthrajhemu.github.io

```
        int number=20;
        switch(number){
        case 10: System.out.println("10");break;
        case 20: System.out.println("20");break;
        case 30: System.out.println("30");break;
        default:System.out.println("Not in 10, 20 or 30");
        }
    }
    }
```

Output : 20

Nested switch Statements

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the
case constants in the inner switch and those in the outer switch.

For example, the following fragment is perfectly valid:

```
switch(count) {
case 1:
        switch(target)
        { // nested switch
                case 0:
                        System.out.println("target is zero");
                        break;
                case 1: // no conflicts with outer switch
                        System.out.println("target is one");
                        break;
        }
break;
case 2: // .........so on.
```

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block
while its controlling expression is true. Here is its general form:

while(*condition*)

https://hemanthrajhemu.github.io

```
{
  // body of loop
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long
as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.

Example:

```
class WhileLoopExample{
        public static void main(String[] args){
                int num=0;
                while(num<=5){
                        System.out.println(""+num);
                        num++;
                }
        }
}
```

do-while

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {

// body of loop

} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.
The do-while loop is especially useful when you process a menu selection, because you will
usually want the body of a menu loop to execute at least once.

```
class Menu {
public static void main(String args[])
{
char choice;
```


https://hemanthrajhemu.github.io

```
do
{
   System.out.println("Help on: ");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. while");
     System.out.println(" 4. do-while");
    System.out.println(" 5. for\n");
    System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');

System.out.println("\n");

switch(choice) {
case '1':
System.out.println("The if:\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" //...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\n");
System.out.println("while(condition) statement;");
break;
case '4':
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while (condition);");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
```


https://hemanthrajhemu.github.io

```
}
}
}
```

```
public class DoWhileExample {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=10);
}
}
```

For:

There are two forms of the for loop.

The first is the traditional form that has been in use since the original version of Java. The second is the newer "for-each" form.

1) Here is the general form of the traditional for statement:

```
for(initialization; condition; iteration)
{
// body
}
```

Ex :

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++)
        sum += nums[i];
```

2) **For-Each Version of the for Loop:**

The general form of the for-each version of the for is shown here:

for(type itr-var : collection) statement-block

Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection(array), one at a time, from beginning to end. The collection being cycled through is specified by collection.

Ex :

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
sum += x;
```


https://hemanthrajhemu.github.io

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names = {"James", "Larry", "Tom", "Lacy"};

      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
```

Iterating Over Multidimensional Arrays

```
class sample {
public static void main(String args[]) {
int sum = 0;
int nums[][] = new int[3][5];
for(int i = 0; i < 3; i++)
for(int j = 0; j < 5; j++)
      nums[i][j] = (i+1)*(j+1);

for(int x[] : nums) {
for(int y : x) {
System.out.println("Value is: " + y);
sum += y;
}
}
System.out.println("Summation: " + sum);
}}
```

Output:
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5

https://hemanthrajhemu.github.io

```
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
```

Jump Statements

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto.

1) Using break to Exit a Loop
2) Using break as a Form of Goto:
   Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain.

```
public class BreakDemo
{
  public static void main(String[] args)
  {
    for (int i = 1; i <= 10; i++)
    {
      if (i == 5)
      {
        break;          // terminate loop if i is 5
      }
      System.out.print(i + " ");
    }
    System.out.println("Thank you.");
  }
}
```

Output 1 2 3 4 Thank you

Using continue

In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to

https://hemanthrajhemu.github.io

the conditional expression. For all three loops, any intermediate code is bypassed.

```
public class ContinueDemo
{
  public static void main(String[] args)
  {
    for (int i = 1; i <= 10; i++)
    {
      if (i % 2 == 0)
      {
        continue;    // skip next statement if i is even
      }
      System.out.println(i + " ");
    }
  }
}
```

```
1 3 5 7 9
```

Break	**Continue**
The break statement results in the termination of the loop, it will come out of the loop and stops further iterations.	The continue statement stops the current execution of the iteration and proceeds to the next iteration
The break statement has two forms: labelled and unlabelled. An unlabelled break statement terminates the innermost switch, for, while, or do-while statement, but a labelled break terminates an outer statement.	The continue statement skips the current iteration of a for, while , or do-while loop. The unlabelled form skips to the end of the innermost loop's body and evaluates the Boolean expression that controls the loop. A labelled continue statement skips the current iteration of an outer loop marked with the given label.
The general form of the labelled break statement is shown here: **break label;**	The general form of the labelled continue statement is shown here: **continue label;**
class Break { public static void main(String args[]) { boolean t = true; **first:** { **second:** { **third:** { System.out.println("Before the break.");	class ContinueLabel { public static void main(String args[]) { **outer:** for (int i=0; i<4; i++) { for(int j=0; j<4; j++) { if(j > i)

https://hemanthrajhemu.github.io

if(t)
break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");}}}
	{
	System.out.println();
	continue outer;
	}
	System.out.print(" " + (i * j));
	}
	}
	System.out.println();
	}
	}

Output: Before the break. This is after second block.	0 0 1 0 2 4 0 3 6 9

Return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

```
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
   if(t)
        return; // return to caller
System.out.println("This won't execute.");
}
}
```

The output from this program is shown here:

Before the return.

Here, return causes execution to return to the Java run-time system, since it is the run-time system that call main( ):

https://hemanthrajhemu.github.io