UNIT-1

UNIT-I

- JAVA BASICS
- History of Java
- Java buzzwords
- Data types
- Variables
- Simple java program
- scope and life time of variables
- Operators, expressions, control statements
- Type conversion and casting
- Arrays
- Classes and objects – concepts of classes, objects
- Constructors, methods
- Access control
- This keyword
- Garbage collection
- Overloading methods and constructors
- Parameter passing
- Recursion
- String handling .

History of Java

- Java started out as a research project.
- Research began in 1991 as the Green Project at Sun Microsystems, Inc.
- Research efforts birthed a new language, OAK. (A tree outside of the window of James Gosling's office at Sun).
- It was developed as an embedded programming language, which would enable embedded system application.
- It was not really created as web programming language.
- Java is available as *jdk* and it is an open source s/w.

History of Java (contd...)

Language was created with 5 main goals:

- It should be object oriented.
- A single representation of a program could be executed on multiple operating systems. (i.e. write once, run anywhere)
- It should fully support network programming.
- It should execute code from remote sources securely.
- It should be easy to use.

- Oak was renamed Java in 1994.

- Now Sun Microsystems is a subsidiary of Oracle Corporation.

James Gosling

Green Team

Java Logo

Versions of Java

Version	Codename	Year	Features Added
JDK 1.0	**Oak**	Jan23,1996	-
JDK 1.1	Rebirth of Java	Feb19,1997	Inner classes JavaBeans, JDBC, RMI, Reflection, AWT.
J2SE 1.2	**Playground**	Dec8, 1998	JIT compiler, Collections, IDL & CORBA, strictfp, Java Plug-in.
J2SE 1.3	**Kestrel**	May8, 2000	HotSpot JVM, JavaSound, Java Naming and Directory Interface, Java Platform Debugger Architecture.
J2SE 1.4	**Merlin**	Feb6, 2002	Preferences API, Logging API, assert, image I/O API, security and cryptography extensions.
J2SE 5.0	**Tiger**	Sep30, 2004	Generics, annotations, Autoboxing, Enumerations, Varargs, Enhanced for each.
JAVA SE 6	**Mustang**	Dec11, 2006	JDBC 4.0, JVM improvements, Improved JAXB, Improved web services, Support for older Win9x versions dropped.
JAVA SE 7	**Dolphin**	July28, 2011	Major updates to Java
JAVA SE 8	-	2012	-

Java Platforms

There are three main platforms for Java:

- **Java SE** (Java Platform, Standard Edition) – runs on desktops and laptops.
- **Java ME** (Java Platform, Micro Edition) – runs on mobile devices such as cell phones.
- **Java EE** (Java Platform, Enterprise Edition) – runs on servers.

Java Terminology

Java Development Kit:

It contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc.

Java Virtual Machine:

An abstract machine architecture specified by the Java Virtual Machine Specification.

It interprets the byte code into the machine code depending upon the underlying OS and hardware combination. JVM is platform **dependent**. (It uses the class libraries, and other supporting files provided in JRE)

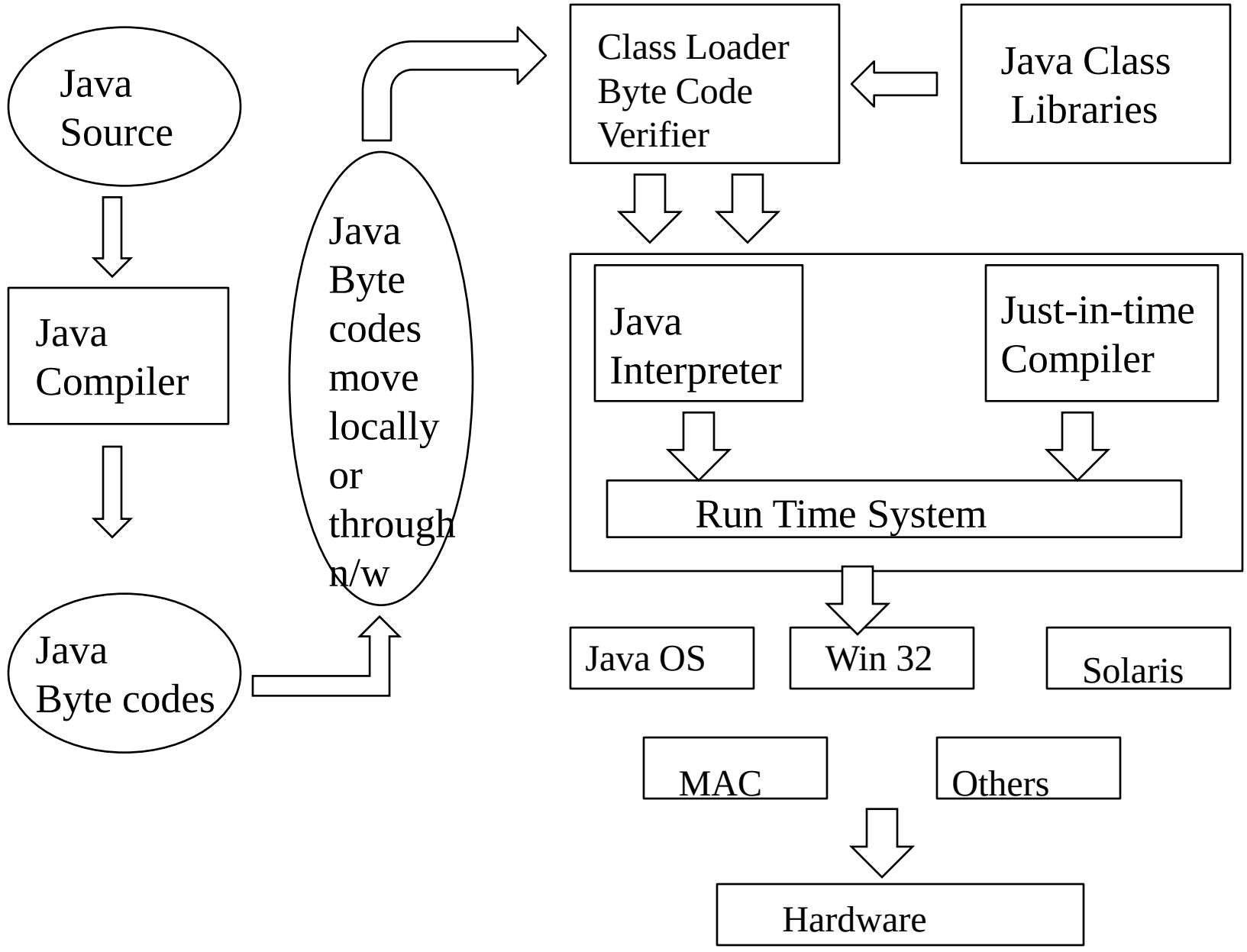
Java Terminology (contd…)

- Java Runtime Environment:

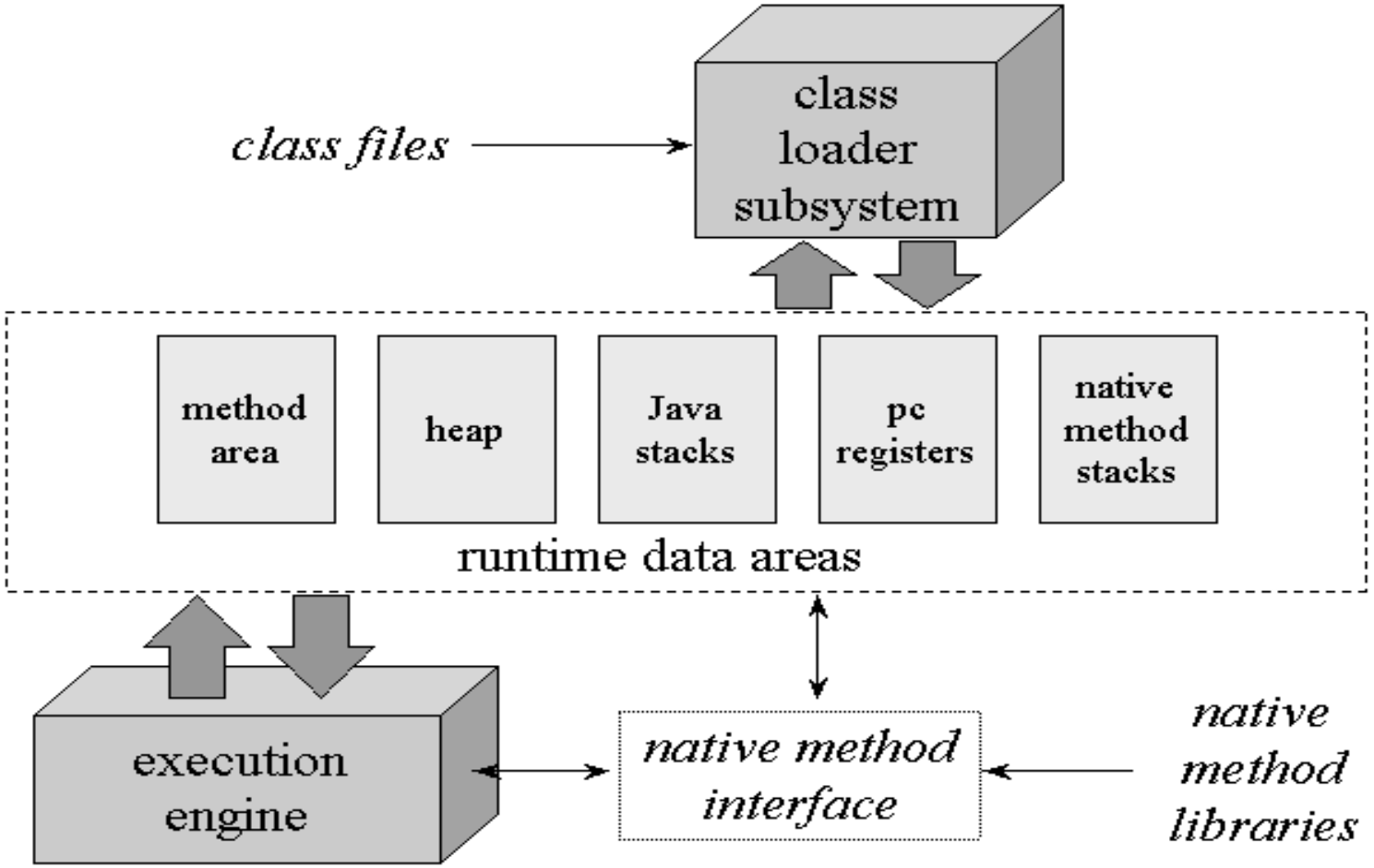
  A runtime environment which implements Java Virtual Machine, and provides all class libraries and other facilities necessary to execute Java programs. This is the software on your computer that actually runs Java programs.

  JRE = JVM + Java Packages Classes (like util, math, lang, awt, swing etc) +runtime libraries.

Java Execution Procedure



The Architecture of the Java Virtual Machine



Java Virtual Machine

- *Class loader subsystem*: A mechanism for loading types (classes and interfaces) given fully qualified names.

- The Java virtual machine organizes the memory it needs to execute a program into several *runtime data areas*.

- Each Java virtual machine also has an *execution engine*: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

Class loader subsystem

- The Java virtual machine contains two kinds of class loaders: a *bootstrap class loader* and *user-defined class loaders*.
- The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application.
- **Loading**: finding and importing the binary data for a type
- **Linking**: performing verification, preparation, and (optionally) resolution
  - Verification: ensuring the correctness of the imported type
  - Preparation: allocating memory for class variables and initializing the memory to default values
  - Resolution: transforming symbolic references from the type into direct references.
- **Initialization**: invoking Java code that initializes class variables to their proper starting values.

When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file.
It places this type information into the method area.

As the program runs, the virtual machine places all objects the program instantiates onto the heap.

As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*.

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).The JVM is an interpreter for byte code.

Binary form of a .class file(partial)

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

```
  0000: cafe babe 0000 002e 001a 0a00 0600 0c09 ..............
  0010: 000d 000e 0800 0f0a 0010 0011 0700 1207 ..............
  0020: 0013 0100 063c 696e 6974 3e01 0003 2829 .....<init>...()
  0030: 5601 0004 436f 6465 0100 046d 6169 6e01 V...Code...main.
  0040: 0016 285b 4c6a 6176 612f 6c61 6e67 2f53 ..([Ljava/lang/S
  0050: 7472 696e 673b 2956 0c00 0700 0807 0014 tring;)V........
  0060: 0c00 1500 1601 000d 4865 6c6c 6f2c 2057 ........Hello, W
  0070: 6f72 6c64 2107 0017 0c00 1800 1901 0005 orld!...........
  0080: 4865 6c6c 6f01 0010 6a61 7661 2f6c 616e Hello...java/lan
  0090: 672f 4f62 6a65 6374 0100 106a 6176 612f g/Object...java/
  00a0: 6c61 6e67 2f53 7973 7465 6d01 0003 6f75 lang/System...ou ...
```

Object Oriented Programming Concepts

- Objects
- Classes
- Data abstraction and Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding

A **class** is collection of objects of similar type or it is a template.

Ex: fruit mango;

class object

Objects are instances of the type class.

The wrapping up of data and functions into a single unit (called class) is known as **encapsulation**. Data encapsulation is the most striking features of a class.

Abstraction refers to the act of representing essential features without including the background details or explanations

Inheritance is the process by which objects of one class acquire the properties of another class. The concept of inheritance provides the **reusability.**

Polymorphism:

It allows the single method to perform different actions based on the parameters.

Dynamic Binding: When a method is called within a program, it associated with the program at run time rather than at compile time is called dynamic binding.

Object-Oriented Languages Types

1.Object-based programming language: It supports

Data encapsulation

Data hiding and access mechanisms

Automatic initialization and clear-up of objects

Operator overloading

Disadvantage : They do not support **inheritance** and **dynamic binding**

Ex: Ada

2.Object-oriented programming languages:

OOP = Object-based + inheritance + dynamic binding

Ex: C++, Java, Smalltalk, Object Pascal

Benefits of OOP

- Through inheritance, we can **eliminate redundant code** and extend the use of existing classes.
- The principle of data hiding helps the programmer to build **secure** programs.
- It is easy to partition the work in a project based on objects.
- Object oriented system easily **upgraded** from small to large systems.
- Software complexity can be easily managed.

Applications of oop

- Real-time systems
- Object-oriented databases
- Neural networks and parallel programming
- Decision support and office automation systems
- CAD/CAM systems

- What is the Difference b/w OO and OB Languages?

  In Object based languages inheritance is not supported so that dynamic polymorphism also not supported.

  E.g. VB,VC++.

- Is C++ partial OOP?

  Yes, C++ is a partial OOP because without using class also we can able to write the program.

- Is Java total OOP or partial OOP?

  Java is a total oop language because with out object orientation we can’t able to write any program.

- Java is a pure oop or not ?

  By default java is not pure object oriented language.

  Java is called as Hybrid language.

  Pure oop languages are “small talk”, ”ruby”, “Eiffel”.

Differences b/w C++ and Java

C++	Java
1. Global variable are supported.	1. No Global variables. Everything must be inside the class only.
2.Multiple inheritance is supported.	2. No direct multiple Inheritance.

C++	Java
3.Constructors and Destructors supported.	3.Java supporting constructors only & instead of destructors garbage collection is supported.
4.In c++ pointers are supported.	4.No pointer arithmetic in Java.
5.C++ supporting ASCII character set.	5. Java supports Uni code Character set.

Features of Java (Java Buzz Words)

- Simple
- Object Oriented
- Compile, Interpreted and High Performance
- Portable
- Reliable
- Secure
- Multithreaded
- Dynamic
- Distributed
- Architecture-Neutral

Java Features

- **Simple**
  - No pointers
  - Automatic garbage collection
  - Rich pre-defined class library

- **Object Oriented**
  - Focus on the data (objects) and methods manipulating the data
  - All methods are associated with objects
  - Potentially better code organization and reuse

Java Features

- **Compile, Interpreted and High Performance**
  - Java compiler generate byte-codes, not native machine code
  - The compiled byte-codes are platform-independent
  - Java byte codes are translated on the fly to machine readable instructions in runtime (Java Virtual Machine)
  - Easy to translate directly into native machine code by using a just-in-time compiler.
- **Portable**
  - Same application runs on all platforms
  - The sizes of the primitive data types are always the same
  - The libraries define portable interfaces

Java Features

- **Reliable/Robust**
  - Extensive compile-time and runtime error checking
  - No pointers but real arrays. Memory corruptions or unauthorized memory accesses are impossible
  - Automatic garbage collection tracks objects usage over time
- **Secure**
  - Java’s robustness features makes java secure.
  - Access restrictions are forced (private, public)

Java Features

- **Multithreaded**
  - It supports multithreaded programming.
  - Need not wait for the application to finish one task before beginning another one.
- **Dynamic**
  - Libraries can freely add new methods and instance variables without any effect on their clients
  - Interfaces promote flexibility and reusability in code by specifying a set of methods an object can perform, but leaves open how these methods should be implemented .

Java Features

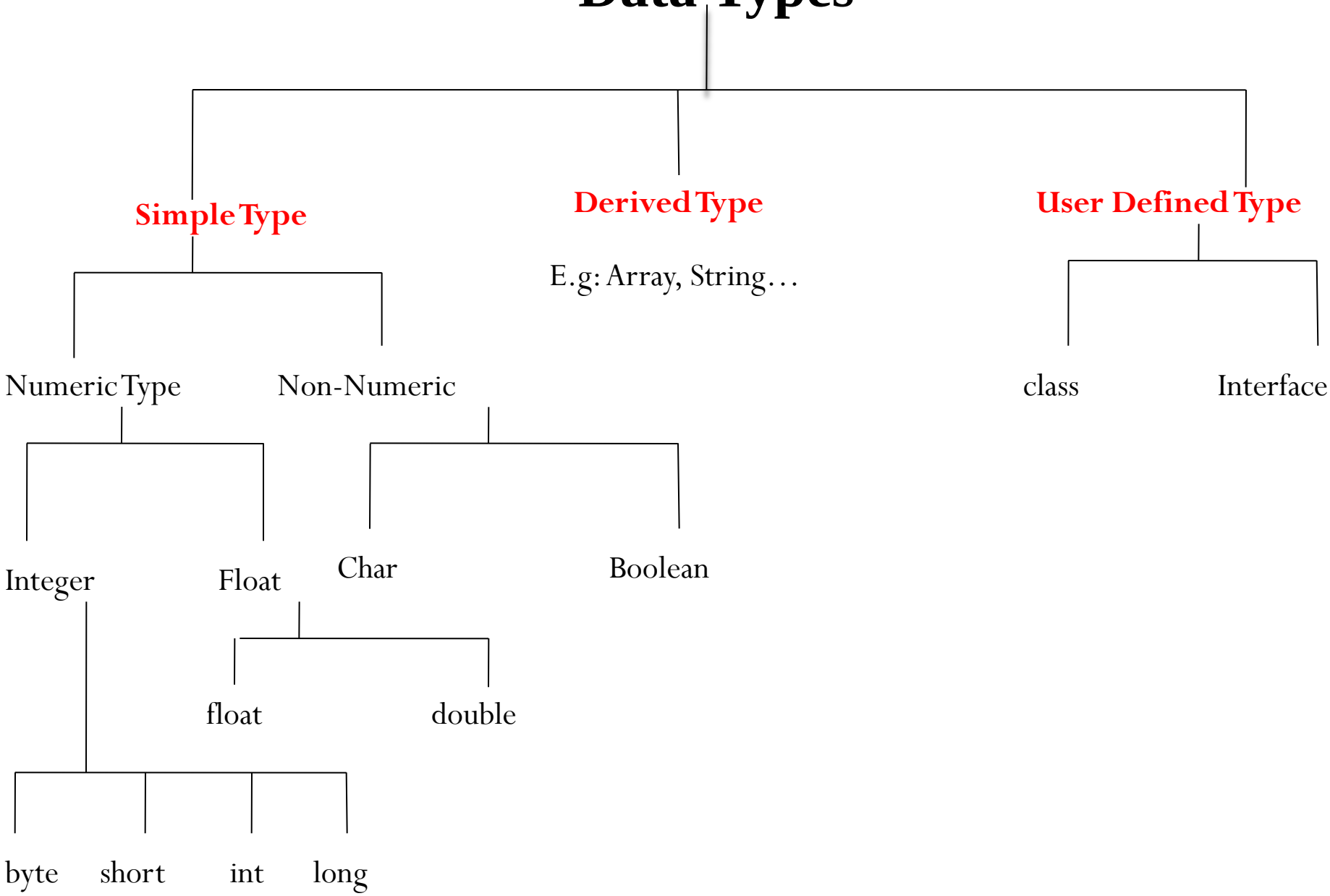
- **Distributed**
  - Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
  - Allows objects on two different computers to execute procedures remotely by using package called *Remote Method Invocation (RMI).*

- **Architecture-Neutral**
  - Goal of java designers is "write once; run anywhere, any time, forever."

Keywords

abstract	continue	goto	package	synchronized
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceOf	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

Data Types


Simple Type
Derived Type
User Defined Type
E.g: Array, String…
Numeric Type
Non-Numeric
class
Interface
Integer
Float
Char
Boolean
float
double
byte
short
int
long

Data Types

- Java Is a Strongly Typed Language
  - Every variable has a type, every expression has a type, and every type is strictly defined.
  - All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
  - There are no automatic conversions of conflicting types as in some languages.
- *For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot.*

Integer Data Types

- Java does not support unsigned, positive-only integers.
- All are signed, positive and negative values.

Byte Data Type

- The smallest integer type is **byte**.
- Variables of type **byte** are especially useful while working with a stream of data from a network or file.
- Byte variables are declared by use of the **byte** keyword.

Ex: byte b, c;

Floating Point Types

- There are two kinds of floating-point types.
- All math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

Boolean Data Type

- It can have only one of two possible values, **true** or **false**.
- This is the type, returned by all relational operators, such as **a < b**.

Character Data Type

- **char** in Java is not the same as **char** in C or C++.
- 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.
- Hence it requires **16 bits**.
- The range of a **char** in java is 0 to 65,536.
- There are no negative **char**s.

Data Types

Name	Width in bits	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038
char	16	0 to 65,536.

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

    System.out.println(" For an Integer ");
    System.out.println("Size is : "+Integer.SIZE);

            int i1 = Integer.MAX_VALUE;
            int i2 = Integer.MIN_VALUE ;

    System.out.println("Max value is : "+i1);
    System.out.println("Min  Value is : "+i2);

     System.out.println(" For an Byte");
    System.out.println("Size is : "+Byte.SIZE);

            byte b1 = Byte.MAX_VALUE;
            byte b2 = Byte.MIN_VALUE ;

    System.out.println("Max value is : "+b1);
    System.out.println("Min  Value is : "+b2);

    System.out.println(" For an Short");
    System.out.println("Size is : "+Short.SIZE);

      short s1 = Short.MAX_VALUE;
      short s2 = Short.MIN_VALUE ;

      System.out.println("Max value is : "+s1);
      System.out.println("Min  Value is : "+s2);


      System.out.println(" For an Long");
      System.out.println("Size is : "+Long.SIZE);

      long l1 = Long.MAX_VALUE;
      long l2 = Long.MIN_VALUE ;

      System.out.println("Max value is : "+l1);
      System.out.println("Min  Value is : "+l2);


       }
}
```

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

            System.out.println(" For an Float");
            System.out.println("Size is : "+Float.SIZE);

            float f1 = Float.MAX_VALUE;
            float f2 = Float.MIN_VALUE ;

            System.out.println("Max value is : "+f1);
            System.out.println("Min  Value is : "+f2);

            System.out.println(" For an Double");
            System.out.println("Size is : "+Double.SIZE);

            double d1 = Double.MAX_VALUE;
            double d2 = Double.MIN_VALUE ;

            System.out.println("Max value is : "+d1);
            System.out.println("Min  Value is : "+d2);
    }
}
```

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

         System.out.println(" For a Char");
         System.out.println("Size is : "+Character.SIZE);

         int f1 = Character.MAX_VALUE;
         long f2 = Character.MIN_VALUE ;

         System.out.println("Max value is : "+f1);
         System.out.println("Min  Value is : "+f2);
   }
}
```

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.

Declaring a Variable

- In Java, all variables must be declared before they can be used.

***type identifier* [= *value*][, *identifier* [= *value*] ...] ;**

Types

- Instance Variable
- Class Variable
- Local Variable
- Parameters

Local variables :

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- Instance variables are created when an object is created with the use of the key word '**new**' and destroyed when the object is destroyed.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class.
- Instance variables have default values.
- Instance variables can be accessed directly by calling the variable name inside the class.
- However within static methods and different class (when instance variables are given accessibility) that should be called using the fully qualified name **ObjectReference.VariableName**

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 static memory.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables.
- Default values are same as instance variables.
- Static variables can be accessed by calling with the class name **ClassName.VariableName**

```
class Variables{
    int i;
    public int j
    static long l=10;
    public static float f;
    char c;
    boolean b;
    void display(int a){
            i=a;
            System.out.println("i value in display: "+i);
    }
    public static void main(String args[]){
            double d=0.0;
            //public double d=0.0; invalid
            Variables v1=new Variables();
            Variables v2=new Variables();
            Variables v3=new Variables();
            v1.display(100);
            v1.i=2;
            v2.i=3;
            v3.i=4;
            System.out.println("i value is: "+v1.i);
            System.out.println("i value is: "+v2.i);
            System.out.println("i value is: "+v3.i);
            System.out.println("i value is: "+v1.j);
            v1.l=20;
            v2.l=30;
            v3.l=40;
            System.out.println("l value is: "+v1.l);
            System.out.println("l value is: "+v2.l);
            System.out.println("l value is: "+v3.l);

            System.out.println("f value is: "+f);
            System.out.println("c value is: "+v1.c);
            System.out.println("b value is: "+v1.b);

            System.out.println("d value is: "+d);
    }
}
```

```
class Variables{
    int i;//instance variable
    public int j ;//instance variable
    static long l=10;//class variable
    public static float f;//class variable
    char c;//instance variable
    boolean b;//instance variable
    void display(int a){
            i=a;
            System.out.println("i value in display: "+i);
    }
    public static void main(String args[]){
            double d=0.0;//local varible
            //public double d=0.0; invalid
            Variables v1=new Variables();
            Variables v2=new Variables();
            Variables v3=new Variables();
            v1.display(100);
            v1.i=2;
            v2.i=3;
            v3.i=4;
            System.out.println("i value is: "+v1.i);
            System.out.println("i value is: "+v2.i);
            System.out.println("i value is: "+v3.i);
            System.out.println("i value is: "+v1.j);
            v1.l=20;
            v2.l=30;
            v3.l=40;
            System.out.println("l value is: "+v1.l);
            System.out.println("l value is: "+v2.l);
            System.out.println("l value is: "+v3.l);

            System.out.println("f value is: "+f);
            System.out.println("c value is: "+v1.c);
            System.out.println("b value is: "+v1.b);

            System.out.println("d value is: "+d);
    }
}
```

Sample Program

```
class HelloWorld {
   public static void main (String args []) {
   System.out.println ("Welcome to Java Programming.....");
   }
}
```

public allows the program to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared. In this case, main ( ) must be declared as public, since it must be called by code outside of its class when the program is started.

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 interpreter before any objects are made.

void states that the main method will not return any value.

main() is called when a Java application begins. In order to run a class, the class must have a main method.

string args[] declares a parameter named args, which is an array of String. In this case, args receives any command-line arguments present when the program is executed.

System is a class which is present in **java.lang** package.

out is a static field present in system class which returns a **PrintStream** object. As out is a static field it can call directly with classname.

println() is a method which present in PrintStream class which can call through the PrintStream object return by static field **out** present in System class to print a line to console.

```
class sample{
	public static void main(String args[]){
		System.out.println("sample:main");
		sample s=new sample();
		s.display();

	}
	void display(){
		System.out.println("display:main");
	}
}
```

The Scope and Lifetime of Variables

Scope

The *scope* of a declared element is the portion of the program where the element is visible.

Lifetime

The *lifetime* of a declared element is the period of time during which it is alive.

The lifetime of the variable can be determined by looking at the context in which they're defined.

- Java allows variables to be declared within any block.
- A block begins with an opening curly brace and ends by a closing curly brace.

- Variables declared inside a scope are not accessible to code outside.
- Scopes can be nested. The outer scope encloses the inner scope.
- Variables declared in the outer scope are visible to the inner scope.
- Variables declared in the inner scope are not visible to the outside scope.

```
public class Scope
{
    public static void main(String args[]){
            int x;  //know to all code within main
            x=10;
            if(x==10){ // starts 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);
    }
}
```

Operators

- Arithmetic Operators
- Bitwise Operators
- Relational Operators
- Boolean Logical Operators

Arithmetic Operators(1)

Operator	Result
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus

Arithmetic Operators(2)

Operator	Result
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
––	Decrement

Example:

```
class IncDec{
        public static void main(String args[]){
         int a = 1;
         int b = 2;
         int c,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);
    }
}
```

```
class OpEquals{
        public static void main(String args[]){
         int a = 1;
         int b = 2;
         int c = 3;
          a += 5;
          b *= 4;
          c += a * b;
          c %= 6;
         System.out.println("a = " + a);
         System.out.println("b = " + b);
         System.out.println("c = " + c);
       }
}
```

Bitwise Operators(1)

- *bitwise operators* can be applied to the integer types, **long**, **int**, **short, byte** and **char.**
- These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left

Bitwise Operators(2)

Operator	Result
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
 00101010      42
&00001111      15
 --------------
 00001010      10
```

```
 00101010      42
^00001111      15
 -------------
 00100101      37
```

```
int a = 64;
a = a << 2;      256
```

```
int a = 32;
a = a >> 2;      8
```

```
11111000      –8
>>1
11111100      –4
```

```
11111111 11111111 11111111 11111111      –1
>>>24
00000000 00000000 00000000 11111111      255
```

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other.
- They determine equality and ordering.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example:

```
public class RelationalOperatorsDemo
{
   public static void main(String args[])
    {
      int x=10,y=5;
      System.out.println("x>y:"+(x>y));
      System.out.println("x<y:"+(x<y));
      System.out.println("x>=y:"+(x>=y));
      System.out.println("x<=y:"+(x<=y));
      System.out.println("x==y:"+(x==y));
      System.out.println("x!=y:"+(x!=y));
    }
}
```

Boolean Logical Operators(1)

- The Boolean logical operators operate only on b**oolean** operands.
- All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT

& //executes both left and right side operands

&& // Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

```
class Test{
        public static void main(String args[]){
                int denom=0,num=20;
                if (denom != 0 && num / denom > 10)
                        System.out.println("Hi");
        }
}
```

Boolean Logical Operators(2)

Operator	Result
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

```
public class TernaryOperatorDemo
{
  public static void main(String args[])
   {
     int x=10,y=12;
     int z;
     z= x > y ? x : y;
     System.out.println("Z="+z);
}
```

Operator Precedence

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

The Precedence of the Java Operators

Expressions

- An expression is a combination of constants (like 10), operators (like +), variables(section of memory) and parentheses (like “(” and “)”) used to calculate a value.

Ex1: x = 1;

Ex2: y = 100 + x;

Ex3: x = (32 - y) / (x + 5)

Control Statements

- Selection Statements: if & switch
- Iteration Statements: for, while and do-while
- Jump Statements: break, continue and return

Selection Statements

```
if (condition)
    statement1;
else
    statement2;
```

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

```
switch (expression)
 {
     case value1:
      // statement sequence
        break;
     case value2:
     // statement sequence
        break;
     …
     case valueN:
     // statement sequence
          break;
     default:
  // default statement sequence
}
```

The *condition* is any expression that returns a **boolean** value.

The *expression* must be of type **byte**, **short**, **int**, or **char**;
Each of the *values* specified in the **case** statements must be of a type compatible with the expression.

Iteration Statements

```
while(condition)
{
    // body of loop
}
```

```
do
 {
     // body of loop
} while (condition);
```

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

Jump Statements

continue; //bypass the followed instructions

break; //exit from the loop

```
label:
- - - -
- - - -
break label;          //it's like goto statement
```

return; //control returns to the caller

Type Conversion and Casting

- **Type conversion**, **typecasting**, refers to different ways of, implicitly or explicitly, changing an entity of one data type into another.

- Types of Conversions:

  1. Widening conversion
  2. Narrowing conversion

Widening Conversion

The widening conversion is permitted in the following cases:

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

- 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.**
- When these two conditions are met, a *widening conversion* takes place.

Ex: char and boolean are not compatible with each other**.**

```
class Widening{
    public static void main(String args[]){
            short s;
            int i1,i2;
            byte b1=10,b2=20;
            s=b1;           //byte to short
            i1=b2;          //byte to int
            System.out.println("byte to short conversion");
            System.out.println(b1+ " " + s);
            System.out.println("byte to int conversion");
            System.out.println(b2+" "+i1);
            char c='a';
            i2=c;           //char to int
            System.out.println("char to int conversion");
            System.out.println(c+" "+i2);
    }
}
```

Narrowing Conversion

- In general, the narrowing primitive conversion can occur in these cases:

  short to byte or char

  char to byte or short

  int to byte, short, or char

  long to byte, short, or char

  float to byte, short, char, int, or long

  double to byte, short, char, int, long, or float

- Narrowing conversion is used to cast the above **incompatible types**.

  (target-type) value

```
public class Narrowing{
   public static void main(String args[]){
          byte b=2;
          int i=257;
          double d=323.142;
          System.out.println("int to byte conversion");
          b= (byte)i;    //int to byte
          System.out.println("i and b values: "+i+" "+b);
          System.out.println("double to int conversion");
          i=(int)d;       //double to int
          System.out.println("d amd i values: "+d+" "+i);
          System.out.println("double to byte conversion");
          b=(byte)d;    //double to byte
          System.out.println("d amd b values: "+d+" "+b);
   }
}
```

Arrays

- An array is a group of like-typed variables that are referred to by a common name.
- The operator **new** is used for dynamic memory allocation.
- One-Dimensional Arrays:

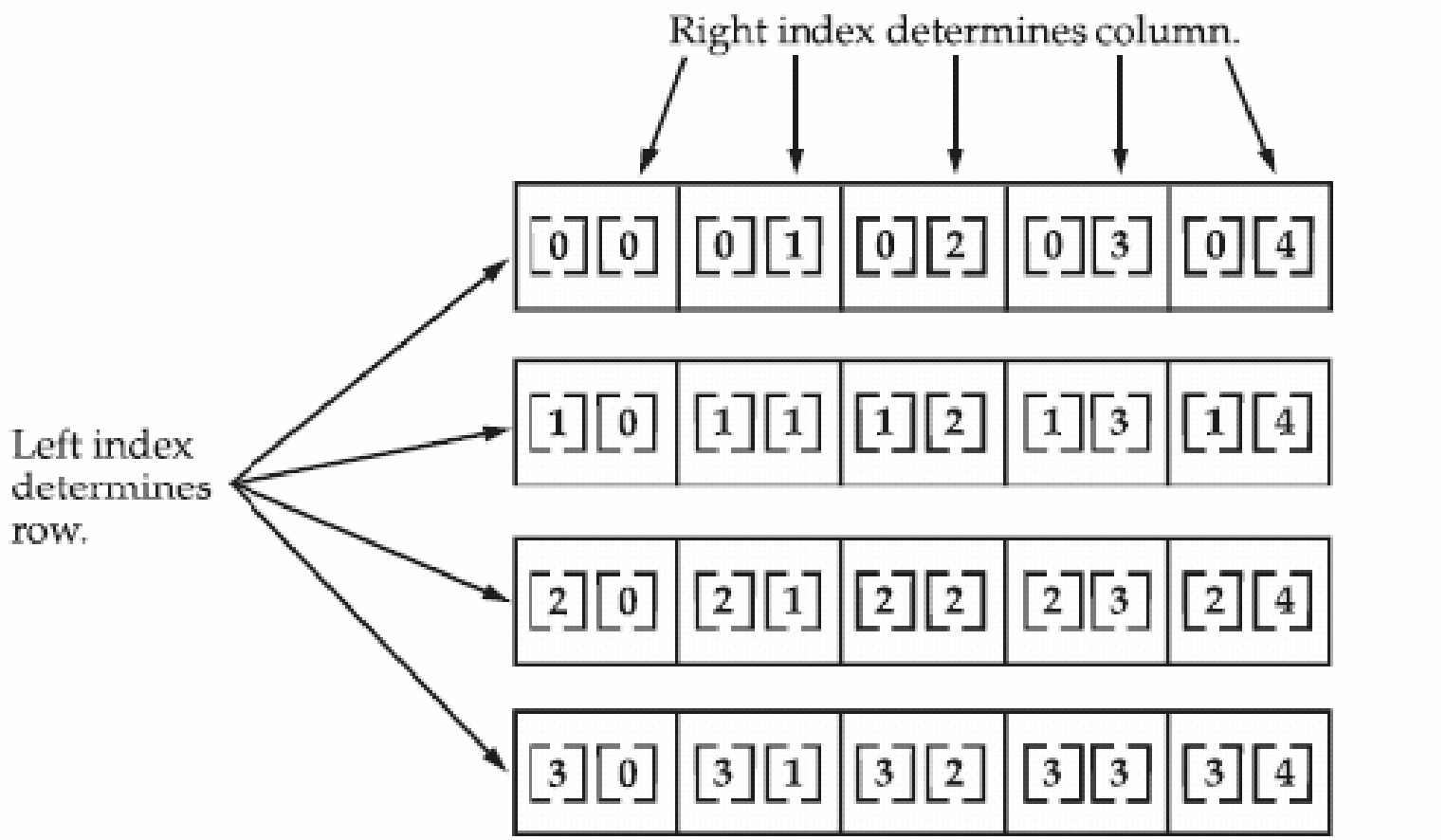
```
type varname[];
varname = new type[size];

type varname[]=new type[size];
```

Ex: int month[];
Ex: month = new int[12];
Ex: int varname[]=new int[size];

Multidimensional Arrays

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



- If **month** is a reference to an array, **month.length** will give you the length of the array.

Initialization:

- int x[] = {1, 2, 3, 4};
- char []c = {‘a’, ‘b’, ‘c’};
- double d[][]= {
  {1.0,2.0,3.0},
  {4.0,5.0,6.0},
  {7.0,8.0,9.0}
  };

Jagged Array:

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

```
import java.util.Scanner;
class ArrayEx{
    public static void main(String args[]){
            Scanner input=new Scanner(System.in);
            int a[]={10,20,30,40,50};
            char []c={'a','b','c','d','e'};
            int b[]=new int[5];
            for(int i=0;i<5;i++){
                    System.out.print(a[i]+" ");
                    System.out.println(c[i]+" ");
            }
            for(int i=0;i<5;i++){
                    b[i]=input.nextInt();
            }
            for(int i=0;i<5;i++){
                    System.out.print(b[i]+" ");
            }
    }
}
```

Concepts of Classes, Objects

General Form of Class

```
class classname {
        type instance-variable1;
        //...
        type instance-variableN;
        static type variable1;
        type methodname1(parameter-list) {
                // body of method
        }
        // ...
        type methodnameN(parameter-list){
                // body of method
        }
}
```

Declaring an Object

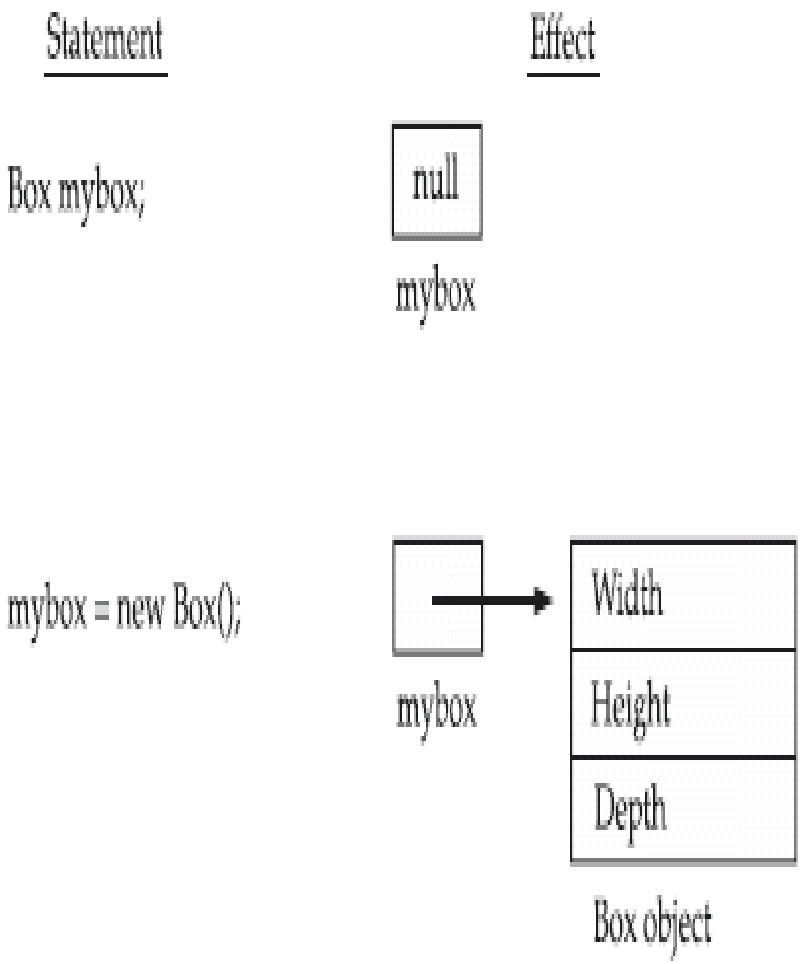
```
class Box{
   double width;
   double height;
   double depth;
}
```

Representation 1:

```
Box mybox;
mybox=new Box();
```

Representation 2:

```
Box mybox=new Box();
```



Representation 3: Assigning Object Reference Variables

```
Box b1 = new Box();
Box b2 = b1;
```

```
package packagename;
import statement;
class classname{
    //instance variables;
    //class or static variables;
    /*methods(parameters){
            //local variables;
            //object creation
            //statements;
    }*/
}
```

```
import java.util.Scanner;
class Demon{
    int i;
    static double d;
public static void main(String args[]){
            char c='a';
            Demon obj=new Demon();
    Scanner s=new Scanner(System.in);
            d=s.nextDouble();
            System.out.println(obj.i);
            System.out.println(d);
            System.out.println(c);
    }
}
```

```
class classname{
    //instance variables;
    //class or static variables;
    /*methods(parameters){
            //local variables;
            //statements;
    }*/
}
class  MainCLass{
public static void main(String args[])
{
            //object creation
            //invoking methods
            //statements
    }
}
```

```
class Test{
    char c='a';
    static float f;
    void display(){
        int i=10;
        System.out.println("Test:display()");
        System.out.println("c value: "+c);
        System.out.println("i value: "+i);
        System.out.println("f value: "+f);
    }
}
class Demo{
    public static void main(String args[]){
    Test t=new Test();
     t.display();
    System.out.println("Demo:main()");
    }
}
```

Constructors and Methods

- A constructor is a special member function whose task is to **initialize** an object immediately upon creation.
- The constructor is **automatically** called immediately after the object is created.
- A constructor has the **same name as the class** in which it resides and is syntactically similar to a method.
- If no constructor in program .System provides its own constructor called as **default** constructor.
- Constructors **doesn't** have any return type.
- A constructor which accepts parameters is called as **parameterized** constructor.
- Constructors can be **overloaded**.

Default Constructor:

- A constructor that accepts no parameters is called Default constructor.
- If not defined, provided by the compiler.
- The default constructor is called whenever an object is created without specifying initial values.

```
Ex:   class Box {
          double width;
          double height;
          double depth;
          Box() {
                width = 10;
                height = 10;
                depth = 10;
              }
      }
  // declare, allocate, and initialize Box objects
```

Box mybox1 = new Box();

// Non parameterized Default Constructor

```
class Test{
    char c='a';
    static float f;
    Test(){
        int i=10;
        System.out.println("Test:Test()");
        System.out.println("c value: "+c);
        System.out.println("i value: "+i);
        System.out.println("f value: "+f);
    }
}
class ConDemo{
    public static void main(String args[]){
    Test t=new Test();
     //t.Test();
System.out.println("ConDemo:main()");
    }
}
```

//Parameterized Constructor

```
class Test{
    int x,y;
    Test(int a, int b){
        x=a;
        y=b;
  System.out.println("x value: "+x);
  System.out.println("y value: "+y);
    }
}
class PConDemo{
public static void main(String args[]){
Test t=new Test(10,20);
System.out.println("PConDemo:main()");
    }
}
```

Methods

General Form:

```
type name(parameter-list) {
        // body of method
}
```

- The type of data returned by a method must be compatible with the return type specified by the method.
- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

```
return value;          //Here, value is the value returned.
```

Ex:

```
double volume(double w, double h, double d) {
        return w*h*d;
}
```

Non parameterized or default Constructor

```
class Box{
    double width;
    double height;
    double depth;

    Box(){
        width=10;
        height=10;
        depth=10;
    }
    double volume(){
    return width*height*depth;
    }
}
```

```
class BoxDemo6{
public static void main(String args[]){

        Box mybox1=new Box();
        Box mybox2=new Box();

        double vol;

        vol=mybox1.volume();
    System.out.println("Volume is: "+vol);

        vol=mybox2.volume();
    System.out.println("Volume is: "+vol);
        }
}
```

Parameterized Constructor

```
class Box{
   double width;
   double height;
   double depth;

Box(double w,double h,double d){
          width=w;
          height=h;
          depth=d;
}
   double volume(){
        return width*height*depth;
   }


}
```

```
class BoxDemo7{
     public static void main(String args[]){

          Box mybox1=new Box(10,20,15);
          Box mybox2=new Box(3,6,9);

          double vol;

          vol=mybox1.volume();

     System.out.println("Volume is: "+vol);

          vol=mybox2.volume();

      System.out.println("Volume is: "+vol);
          }
 }
```

Access Control

default :

- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

private:

- A private member is accessible only to the class in which it is defined.
- Use **private** keyword to create private members.

public:

- Any class, in any package has access to a class's public members.
- To declare a public member, use the keyword **public**.

protected:

- Allows the class itself, subclasses, and all classes in the same package to access the members.
- To declare a protected member, use the keyword **protected**.

//Example for access control

```
class Test {
   int a;            // default access
   public int b;     // public access
   private int c;    // private access
   /*protected applies only
  when inheritance is involved*/

   // methods to access c

   void setc(int i){
         c = i;      // set c's value

   }
   int getc() {
         return c;   // get c's value
   }
}
```

```
class AccessTest {

         public static void main(String args[]) {

         Test ob = new Test();

// These are OK, a and b may be accessed directly
         ob.a = 10;
         ob.b = 20;

// This is not OK and will cause an error
         //ob.c = 100; // Error!

// You must access c through its methods
         ob.setc(100); // OK

System.out.println(ob.a + " " +ob.b + " " + ob.getc());
         }
}
```

Method & Constructor Overloading

- Defining two or more methods within the same class that share the same name is called **method overloading**.

- Java uses the type and/or number of arguments to determine which version of the overloaded method to call.

- Constructors can also be **overloaded** in the same way as method overloading.

```
class OverloadDemo {
    void test() {
            System.out.println("No parameters");
    }
    void test(int a) {
            System.out.println("a: " + a);
    }
    void test(int a, int b) {
            System.out.println("a and b: " + a + " " + b);
    }
    double test(double a) {
            System.out.println("double a: " + a);
    return a*a;
    }
}
```

//method overloading

```
class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                double result;
                ob.test();
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.25);
                System.out.println("Result of ob.test(123.25): " + result);
}       }
```

```
//Constructor Overloading
class CDemo{
    int  value1;
    int  value2;
    CDemo(){
            value1 = 10;
            value2 = 20;
    System.out.println("Inside 1st
                        Constructor");
    }
    CDemo(int a){
            value1 = a;
            System.out.println("Inside 2nd Constructor");
    }
    CDemo(int a,int b){
            value1 = a;
            value2 = b;
            System.out.println("Inside 3rd Constructor");
    }
    public void display(){
            System.out.println("Value1 === "+value1);
            System.out.println("Value2 === "+value2);
    }
```

```
public static void main(String args[]){
                        CDemo d1 = new CDemo();
                        CDemo d2 = new CDemo(30);
                        CDemo d3 = new CDemo(30,40);
                        d1.display();
                        d2.display();
                        d3.display();
                }
}
```

this

- In java, it is illegal to declare two local variables with the same name inside the same or enclosing scopes.
- But you can have formal parameters to methods, which overlap with the names of the class' instance variables.
- **this** keyword is used to refer to the **current** object.
- **this** can be used to resolve any name collisions that might occur between instance variables and formal variables.
- When a formal variable has the same name as an instance variable, the formal variable **hides** the instance variable.
- Also used in **method chaining** and **constructor chaining**.

```
// instance and formal variables are different
class Box{
    double w=5,h=5,d=5;
    Box(double w1,double h1,double d1){
            w=w1;
            h=h1;
            d=d1;
    }
    double volume(){
            return w*h*d;
    }
}
class BoxTest1{
    public static void main(String args[]){
            Box b=new Box(1,2,3);
            System.out.println("Volume is: "+b.volume());
    }
}
```

Output:
Volume is:6.0

```
// instance and formal variables are same
class Box{
    double w=5,h=5,d=5;
    Box(double w,double h,double d){
            w=w;
            h=h;
            d=d;
    }
    double volume(){
            return w*h*d;
    }
}
class BoxTest2{
    public static void main(String args[]){
            Box b=new Box(1,2,3);
            System.out.println("Volume is: "+b.volume());
    }
}
```

Output:

Volume is:125.0

```
//'this' hides the instance variables
class Box{
    double w=5,h=5,d=5;
    Box(double w,double h,double d){
            this.w=w;
            this.h=h;
            this.d=d;
}
    double volume(){
            return w*h*d;
    }
}
class BoxTest2{
    public static void main(String args[]){
            Box b=new Box(1,2,3);
            System.out.println("Volume is: "+b.volume());
    }
}
```

Output:
Volume is:6.0

```
class Fchain{                    // method chaining
        int a,b;
        Fchain setValue(int x,int y){
                a=x;
                b=y;
                return this;
        }
        Fchain disp(){
                System.out.println("a value is:"+a);
                System.out.println("b value is:"+b);
                return this;
        }
}
class FchainDemo{
        public static void main(String args[]){
        Fchain f1=new Fchain();
        f1.setValue(10,20).disp().setValue(11,22).disp();
        }
}
```

```
//Constructor Chaining
class Test{
        int a,b,c,d;
        Test(int x,int y){
                a=x;
                b=y;
        }
        Test(int x,int y,int z){
                this(x,y);
                c=z;
        }
        Test(int p,int q,int r,int s){
                this(p,q,r);
                d=s;
        }
        void disp(){
   System.out.println(a+" "+b+" "+c+" "+d);
        }
}
```

```
class TestDemo{
        public static void main(String args[]){
                Test t1=new Test(10,20,30,40);
                t1.disp();
                }
}
```

Parameter Passing

- The **call-by-value** copies the *value* of a actual parameter into the formal parameter of the method.
- In this method, changes made to the formal parameter of the method have no effect on the actual parameter.
- In **call-by-reference**, a reference to an actual parameter (not the value of the argument) is passed to the formal parameter.
- In this method, changes made to the actual parameter will affect the actual parameter used to call the method.

```
// Simple types are passed by value.
class Test {
    void meth(int i, int j) {
            i *= 2;
            j /= 2;
    }
}
class CallByValue {
    public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a and b before call: " +a + " " + b);
    ob.meth(a, b);
    System.out.println("a and b after call: " +a + " " + b);
    }
}
```

```
// Objects are passed by reference.
class Test {
    int a, b;
    Test(int i, int j) {
            a = i;
            b = j;
    }
    void meth(Test o) {                    // pass an object
            o.a *= 2;
            o.b /= 2;
    }
}
class CallByRef {
    public static void main(String args[]) {
            Test ob = new Test(15, 20);
            System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);
            ob.meth(ob);
            System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
}           }
```

```
class Box{     //call by reference
    double width,height,depth;
    Box(Box ob) {
            width = ob.width;
            height = ob.height;
            depth = ob.depth;
    }
    Box(double w, double h, double d){
            width = w;
            height = h;
            depth = d;
    }
    Box(){
            width = -1;
            height = -1;
            depth = -1;
     }
    Box(double len) {
            width = height = depth = len;
    }
    double volume() {
            return width * height * depth;
    }
}
```

```
class OverloadCons2 {
            public static void main(String args[]) {
            Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box();
            Box mycube = new Box(7);
            Box myclone = new Box(mybox1);
            double vol;

            vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

            vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

            vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

            vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
            }
}
```

Recursion

- Recursion is the process of defining something in terms of itself.

- A method that calls itself is said to be *recursive*.

```
class Factorial{
    int fact(int n){
            int result;
            if(n==1)
                    return 1;
            else
                    result = fact(n-1) * n;
                    return result;
    }
}
class Recursion {
            public static void main(String args[]) {
                    Factorial f = new Factorial();
                    System.out.println("Factorial of 3 is " + f.fact(3));
                    System.out.println("Factorial of 4 is " + f.fact(4));
                    System.out.println("Factorial of 5 is " + f.fact(5));
            }
}
```

Garbage Collection

- Garbage collection done automatically in java.
- When no reference to an object exist, that object is assumed to be no loner needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs at regular intervals during the execution of your program.
- We can run garbage collection on demand by calling the **gc()** method.
- **public static void gc():** Initiates the garbage collection.

System.gc();

```
public class GarbageCollector{
    public static void main(String[] args) {
        int SIZE = 200;
        StringBuffer s;
        for (int i = 0; i < SIZE; i++) {
        }
        System.out.println("Garbage Collection started explicitly.");
        System.gc();
    }
}
```

finalize( ) method

- Sometimes an object will need to perform some action when it is destroyed.

  Ex:

  If an object is holding some non-java resource such as a file, then you might want to make sure these resources are freed before an object is destroyed.

- To handle such situations, Java provides a mechanism called *finalization*.

- **The finalize( ) method has this general form:**

```
protected void finalize( ){
        // finalization code here
}
```

String Handling

Types of string handling classes:

- String

  Once an object is created, no modifications can be done on that.

- StringBuffer

  Modifications can be allowed on created object.

- StringTokenizer

  Used to divide the string into substrings based on tokens.

String and SringBuffer classes are available in java.lang package where as StringTokenizer class is available in java.util package.

String

- In java a string is a sequence of characters. They are objects of type String.
- Once a String object has been created, we can not change the characters that comprise in the string.
- Strings are unchangeable once they are created so they are called as immutable.
- You can still perform all types of string operations. But, a new **String** object is created that contains the modifications. The original string is left unchanged.
- To get changeable strings use the class called StringBuffer.
- String and StringBuffer classes are declared final, so there cannot be subclasses of these classes.
- String class is defined in java.lang package, so these are available to all programmers automatically.

String Constructors

- String s= new String();
- String(char chars[])
- String(char chars[], int startIndex, int numChars)
- String(String strobj)
- String str=new String(“SNIST”);
- String str[]=new String[size];

String Constructors

- String s= new String(); //To create an empty String call the default constructor.

- String(char chars[]) //To create a string initialized by an array of characters.

  String str = "abcd"; is **equivalent** to

  char chars[]={'a','b','c','d'};

  String s=new String(chars);

- String(char chars[], int startIndex, int numChars) //To create a string by specifying positions from an array of characters

  char chars[]={'a','b','c','d','e','f'};

  String s=new String(chars,2,3); //This initializes **s** with characters "cde".

- String(String str); //Construct a string object by passing another string object.

  String str = "abcd";

  String str2 = new String(str);

- String(byte asciiChars[ ]) // Construct a string from subset of byte array.
- String(byte asciiChars[ ], int startIndex, int numChars)

```
// Construct one String from another.

class MakeString {
    public static void main(String args[]) {
            char c[] = {'J', 'a', 'v', 'a'};
            String s1 = new String(c);
            String s2 = new String(s1);
            System.out.println(s1);
            System.out.println(s2);
    }
}
```

Output:

```
Java
Java
```

```
// Construct string from subset of char array.

class SubStringCons {
    public static void main(String args[]) {
        byte ascii[] = {65, 66, 67, 68, 69, 70 };
            String s1 = new String(ascii);
            System.out.println(s1);
     String s2 = new String(ascii, 2, 3);
            System.out.println(s2);
    }
}
```

Output:

```
ABCDEF
CDE
```

String Length

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());        //it returns 3.
```

- **Special String Operations**
  - String Concatenation
  - String Concatenation with Other Data Types
  - String Conversion and toString( )
- **Character Extraction**
  - charAt( )
  - getChars( )
  - getBytes( )
  - toCharArray( )
- **String Comparison**
  - equals( ) and equalsIgnoreCase( )
  - regionMatches( ), startsWith( ) and endsWith( )
  - equals( ) Versus ==
  - compareTo( )
- **Searching Strings**
- **Modifying a String**
  - substring( )
  - concat( )
  - replace( )
  - trim( )
- **Changing the Case of Characters Within a String**

Special String Operations

String Concatenation

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);                    // He is 9 years old.
```

String Concatenation with Other Data Types

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);                    // He is 9 years old.
```

```
String s = "four: " + 2 + 2;
System.out.println(s);
        //This fragment displays four: 22  rather than the four: 4
```

```
String s = "four: " + (2 + 2);
        //Now s contains the string “four: 4”.
```

Special String Operations

Box's toString() method is automatically invoked when a Box object is used in a concatenation expression or in a call to println( ).

String Conversion using toString( )

```
// Override toString() for Box class.
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
}
    public String toString() {
    return "Dimensions are " + width + " by "
    +depth + " by " + height + ".";
    }
}
```

```
class toStringDemo {
public static void main(String args[]) {
            Box b = new Box(10, 12, 14);
            String s = "Box b: " + b;
            // concatenate Box object
            System.out.println(b);
            // convert Box to string
            System.out.println(s);
            }
}
```

The output of this program is shown here:

Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

char **charAt**(int *where*)

//To extract a single character from a **String**

```
char ch;
ch = "abc".charAt(1);
//assigns the value “b” to ch.
```

void **getChars**(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

//to extract more than one character at a time.

byte[ ] **getBytes**( )

//to store the characters in an array of bytes and it uses the default character-to-byte conversions provided by the platform.

```
//String source=“hi”+”how”+”are”+”you.”
//byte buf[] = source.getBytes();
```

char[ ] **toCharArray**( )

//to convert all the characters in a String object into a character array. It returns an array of characters for the entire string.

```
class getCharsDemo {
public static void main(String args[]) {
    String s = "This is a demo of the
            getChars method.";
    int start = 10;
    int end = 14;
    char buf[] = new char[end - start];
    s.getChars(start, end, buf, 0);
    System.out.println (buf);
    }
}
```

Here is the output of this program: demo

```
class GetBytesDemo{
    public static void main(String[] args){
        String str = "abc“ + ”ABC”;
            byte[] b = str.getBytes();
            //char[] c=str.toCharArray();
            System.out.println(str);
            for(int i=0;i<b.length;i++){
                System.out.print(b[i]+"  ");
                //System.out.print(c[i]+"  ");
            }
    }
}
```

Output:

97	**98**	**99**	**65**	**66**	**67**
//a	b	c	A	B	C

String Comparison

- boolean **equals**(Object str)

  //To compare two strings for equality. It returns true if the strings contain the same characters in the same order, and false otherwise.

- boolean **equalsIgnoreCase**(String str)

  //To perform a comparison that ignores case differences.

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
        public static void main(String args[]) {
                String s1 = "Hello";
                String s2 = "Hello";
                String s3 = "Good-bye";
                String s4 = "HELLO";
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s1.equals(s4));

        System.out.println(s1.equalsIgnoreCase(s4));
                }
}
```

The output from the program is shown here:

```
true
false
false
true
```

String Comparison

boolean **regionMatches**(int startIndex, String str2, int str2StartIndex, int numChars)

boolean **regionMatches**(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

//The regionMatches( ) method compares a specific region inside a string with another specific region in another string.

//startIndex specifies the index at which the region begins within the invoking String object.

//The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex.

//The length of the substring being compared is passed in numChars.

String Comparison

```
class RegionTest{
        public static void main(String args[]){
                String str1 = "This is Test";
                String str2 = "THIS IS TEST";
                if(str1.regionMatches(5,str2,5,3)) {
                        // Case, pos1,secdString,pos1,len
                        System.out.println("Strings are Equal");
                 }
                else{
                        System.out.println("Strings are NOT Equal");
                }
        }
}
```

Output:

Strings are NOT Equal

String Comparison

boolean **startsWith**(String str) //to determine whether a given String begins with a specified string.

boolean **endsWith**(String str) // to determine whether the String in question ends with a specified string.

Ex: "Football".endsWith("ball") and "Football".startsWith("Foot") are both **true**.

boolean startsWith(String str, int startIndex) //to specifies the index into the invoking string at which point the search will begin.

Ex: "Football".startsWith("ball", 4) returns **true**.

equals( ) Versus ==

// It compares the characters inside a String object

//To compare two object references to see whether they refer to the same instance.

String Comparison

```
// equals() vs ==
class EqualsNotEqualTo {
   public static void main(String args[]) {
          String s1 = "Hello";
          String s2 = new String(s1);
          //String s2 = s1;
          System.out.println(s1.equals(s2));
          System.out.println( s1 == s2);
   }
}
```

Output:

true
false

String Comparison

int **compareTo**(String str)

Value	Meaning
Less than zero	The invoking string is less than str.
Greater than zero	The invoking string is greater than str.
Zero	The two strings are equal.

int compareToIgnoreCase(String str)

String Comparison

```
class SortString {
    static String arr[] = {
            "Now", "is", "the", "time", "for", "all", "good", "men",
            "to", "come", "to", "the", "aid", "of", "their", "country“};
    public static void main(String args[]) {
            for(int j = 0; j < arr.length; j++) {
                    for(int i = j + 1; i < arr.length; i++) {
                            if(arr[i].compareTo(arr[j]) < 0) {
                                    String t = arr[j];
                                    arr[j] = arr[i];
                                    arr[i] = t;
                            }
                    }
                    System.out.println(arr[j]);
            }
    }
}
```

The output of this program is the list of words:

Now	aid	all	come	country	for	good	is	men	of
the	the	their	time	to	to				