Object Oriented Programming Concepts through JAVA
Course Material

B. TECH II YEAR - II SEM
(2019-20)

Prepared By
T. Sampath Kumar
K. Ravi Chythanya
S. Tharun Reddy

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SR ENGINEERING COLLEGE
Approved by AICTE, New Delhi | Affiliated to JNTUH | Accredited by NAAC 'A' Grade
Ananthasagar (V), Hasanparthy (M), WARANGAL - 506 371
(An Autonomous Institution)

UNIT I

Object-oriented thinking- A way of viewing world – Agents and Communities, messages and methods, Responsibilities, Classes and Instances, Class Hierarchies- Inheritance, Method binding, Overriding and Exceptions, Summary of Object-Oriented concepts. Java buzzwords, An Overview of Java, Data types, Variables and Arrays, operators, expressions, control statements, Introducing classes, Methods and Classes, String handling.

UNIT II

Inheritance– Inheritance concept, Inheritance basics, Member access, Constructors, Creating Multilevel hierarchy, super uses, using final with inheritance, Polymorphism-ad hoc polymorphism, pure polymorphism, method overriding, abstract classes, Object class, forms of inheritance- specialization, specification, construction, extension, limitation, combination, benefits of inheritance.

Packages- Defining a Package, CLASSPATH, Access protection, importing packages. Command Line Arguments.

UNIT III

Interfaces- defining an interface, implementing interfaces, Nested interfaces, applying interfaces, variables in interfaces and extending interfaces.

Exception handling - Fundamentals of exception handling, Exception types, Termination or resumptive models, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built- in exceptions, creating own exception sub classes.

UNIT IV

Stream based I/O(java.io) – The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, Random access file operations, The Console class, Serialization, Enumerations, auto boxing, generics.

Multithreading- Differences between thread-based multitasking and process-based multitasking, Java thread model, creating threads, thread priorities, synchronizing threads, inter thread communication.

UNIT V

The Collections Framework (java.util)- Collections overview, Collection Interfaces, The Collection classes- Array List, Linked List, Hash Set, Tree Set, Priority Queue, Array Deque. Accessing a Collection via an Iterator, Using an Iterator, The For-Each alternative, Map Interfaces and Classes, Comparators, Collection algorithms, Arrays, The Legacy Classes and Interfaces- Dictionary, Hash table, Properties, Stack, Vector More Utility classes, String Tokenizer, Bit Set, Date, Calendar, Random, Formatter, Scanner

UNIT VI

GUI Programming with Swing:

Event Handling- The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes, Inner classes, Anonymous Inner classes.

A Simple Swing Application, **Applets –** Applets and HTML, Security Issues, Applets and Applications, passing parameters to applets. Creating a Swing Applet, Painting in Swing, A Paint example, Exploring Swing Controls- JLabel and Image Icon, JText Field, The Swing Buttons- JButton, JToggle Button, JCheck Box, JRadio Button, JTabbed Pane, JScroll Pane, JList, JCombo Box, Swing Menus, Dialogs. Layout Mangers.

TEXT BOOK:

1. Java-The Complete Reference 9th Edition, Hebert Schlidt, McGraw Hill Education (India) Pvt. Ltd.
2. Understanding Object-Oriented Programming with Java, updated edition, T. Budd, Pearson Education.

REFERENCE BOOKS:

1. An Introduction to Programming and OO Design using Java, J. Nino and F.A. Hosch, John Wiley & Sons.
2. Introduction to Java Programming, Y. Daniel Liang, Pearson Education.
3. Object Oriented Programming through Java, P. Radha Krishna, University Press.
4. Programming in Java, S. Malhotra, S. Chudhary, 2nd Edition, Oxfoord Univ. Press.
5. Java Programming and Object-Oriented application development, R.A. Johnson, Cengage Learning.

Lecture No.	Topic	Delivery Method/ Activity
	UNIT – I	
L1	**Object- oriented Thinking**-A way of viewing world- Agents and Communities, messages and methods, Responsibilities	Chalk & Talk/ PPT
L2	Classes and Instances	Chalk & Talk/ PPT
L3	Class Hierarchies- Inheritance, Method binding	Chalk & Talk/PPT
L4	Overriding and Exceptions, Summary of Object-Oriented Concepts	Chalk & Talk/PPT
L5	Java Buzz Words, An Overview of Java, Data Types, Variables and Arrays	Chalk & Talk/PPT
L6	Operators, expressions, control Statements	Chalk & Talk/PPT
L7	Introducing classes, Methods and classes	Chalk & Talk/PPT
L8	String handling, Structured Programming Vs OOP Concepts	Chalk & Talk, **Activity:** Group Discussion

Topic Learning Outcomes:

At the end of this Unit, the students are able to:

1) Identify the basic concepts of Object-Oriented Concepts.
2) Demonstrate the concept of classes and its hierarchy.
3) Illustrate the Programs which include the basics of the programming language.
4) Differentiate the Structured Programming and Object-Oriented Programming.
5) Develop the Java Programs using Object to bind the real-world Objects.

UNIT-I

1.1. Object-Oriented Thinking:

1.5.1 A Way of Viewing World:

To illustrate the major ideas in object-oriented programming, let us consider how we might go about handling a real-world situation and then ask how we could make the computer more closely model the techniques employed. Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who happens to be named Flora), tell her the variety and quantity of flowers I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.

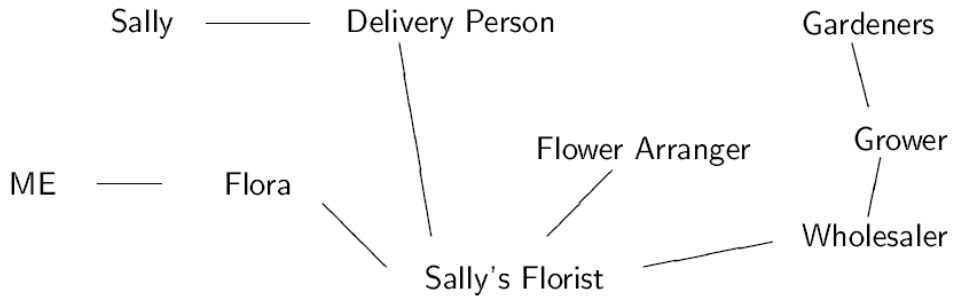


Fig 1. 1: The community of Agents helping me

1.5.2 Agents and Communities:

At the risk of belabouring a point, let me emphasize that the mechanism I used to solve my problem was to find an appropriate agent (namely, Flora) and to pass to her a message containing my request. It is the responsibility of Flora to satisfy my request. There is some method - some algorithm or set of operations - used by Flora to do this. I do not need to know the particular method she will use to satisfy my request; indeed, often I do not want to know the details. This information is usually hidden from my inspection.

If I investigated however, I might discover that Flora delivers a slightly different message to another florist in my friend's city. That florist, in turn, perhaps has a subordinate who makes the flower arrangement. The florist then passes the flowers, along with yet another message, to a delivery person, and so on. Earlier, the florist in Sally's city had obtained her flowers from a flower wholesaler who, in turn, had interactions with the flower growers, each of whom had to manage a team of gardeners.

> *An object-oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.*

1.5.3 Messages and Methods:

The chain reaction that ultimately resulted in the solution to my program began with my request to Flora. This request lead to other requests, which lead to still more

requests, until my flowers ultimately reached my friend. We see, therefore, that members of this community interact with each other by making requests. So, our next principle of object-oriented problem solving is the vehicle by which activities are initiated:

> *Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.*

1.5.4 Responsibilities:

A fundamental concept in object-oriented programming is to describe behaviour in terms of responsibilities. My request for action indicates only the desired outcome (flowers for my friend). Flora is free to pursue any technique that achieves the desired objective and is not hampered by interference on my part.

By discussing a problem in terms of responsibilities we increase the level of abstraction. This permits greater independence between objects, a critical factor in solving complex problems. The entire collection of responsibilities associated with an object is often described by the term protocol. A traditional program often operates by acting on data structures, for example changing fields in an array or record. In contrast, an object-oriented program requests data structures (that is, objects) to perform a service. This difference between viewing software in traditional, structured terms and viewing it from an object-oriented perspective can be summarized by a twist on a well-known quote:

> *Ask not what you can do to your data structures,*
> *but rather ask what your data structures can do for you.*

1.5.5 Classes and Instances:

Let us incorporate these notions into our next principle of object-oriented programming:

> All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

1.5.6 Class Hierarchies – Inheritance:

I have more information about Flora - not necessarily because she is a florist but because she is a shopkeeper. I know, for example, that I probably will be asked for money as part of the transaction, and that in return for payment I will be given a receipt. These actions are true of grocers, stationers, and other shopkeepers. Since the category ***Florist*** is a more specialized form of the category ***Shopkeeper***, any knowledge I have of ***Shopkeepers*** is also true of ***Florists*** and hence of Flora.

One way to think about how I have organized my knowledge of Flora is in terms of a hierarchy of categories (see Fig 1.2). Flora is a ***Florist***, but ***Florist*** is a specialized form of ***Shopkeeper***. Furthermore, a ***Shopkeeper*** is also a ***Human***; so, I know, for example, that Flora is probably bipedal. A ***Human*** is a ***Mammal*** (therefore they nurse their young and have hair), and a ***Mammal*** is an ***Animal*** (therefore it breathes oxygen),

and an ***Animal*** is a ***Material*** Object (therefore it has mass and weight). Thus, quite a lot of knowledge that I have that is applicable to Flora is not directly associated with her, or even with her category ***Florist***.

The principle that knowledge of a more general category is also applicable to a more specific category is called *inheritance*. We say that the class ***Florist*** will inherit attributes of the class (or category) ***Shopkeeper***.

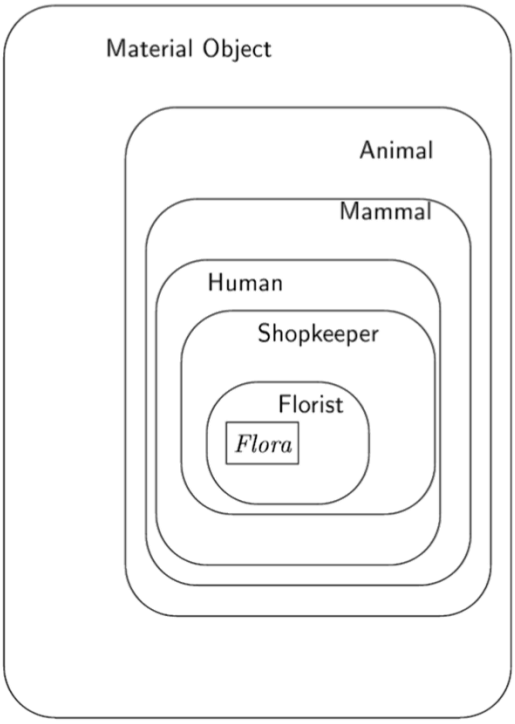


Fig 1. 2: The categories surrounding Flora

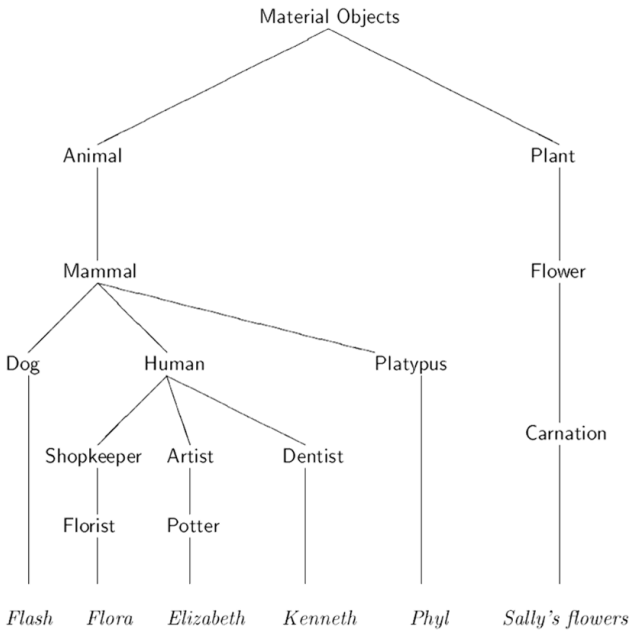


Fig 1. 3: A class hierarchy for various ***Material*** objects

There is an alternative graphical technique often used to illustrate this relationship, particularly when there are many individuals with differing lineages. This technique shows classes listed in a hierarchical tree-like structure, with more abstract classes (such as ***Material, Object*** or ***Animal***) listed near the top of the tree, and more specific classes, and finally individuals, are listed near the bottom. Fig 1.3 shows this class hierarchy for Flora.

> Classes can be organized into a hierarchical inheritance structure. A child class (or subclass) will inherit attributes from a parent class higher in the tree. An abstract parent class is a class (such as Mammal) for which there are no direct instances; it is used only to create subclasses.

1.5.7 Method Binding, Overriding and Exceptions:

> The search for a method to invoke in response to a given message begins with the class of the receiver. If no appropriate method is found, the search is conducted in the parent class of this class. The search continues up the parent class chain until either a method is found or the parent class chain is exhausted. In the former case the method is executed; in the latter case, an error message is issued. If methods with the same name can be found higher in the class hierarchy, the method executed is said to override the inherited behaviour.

1.2. Features of OOP:

Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc. Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. Smalltalk is considered as the first truly object-oriented programming language.

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

- Object
- Class
- Encapsulation
- Abstraction
- Polymorphism
- Inheritance
- Message Passing

Object:

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

Class:

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

Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is 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. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever. You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission.

Abstraction:

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole. Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. In java, we use abstract class and interface to achieve abstraction.

Polymorphism:

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. 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. In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

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. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth and mammary glands. This is known as a subclass of animals, where animals are referred to as mammals' superclass. 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. Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization. This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

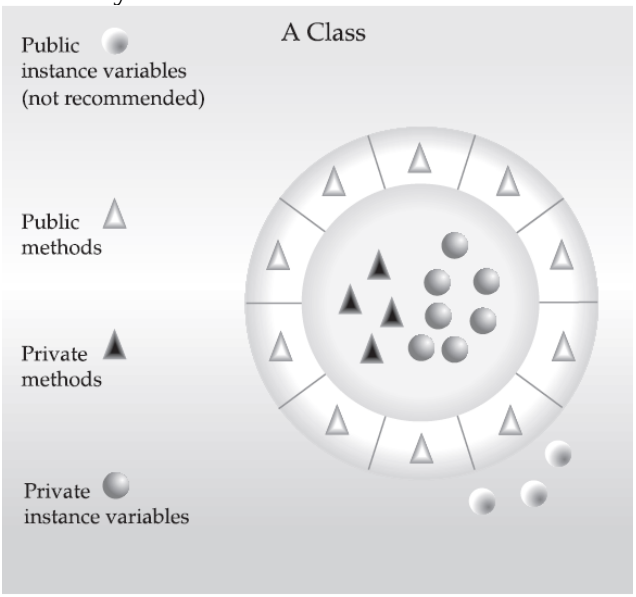


Fig 1. 4: Encapsulation: public methods can be used to protect private data.

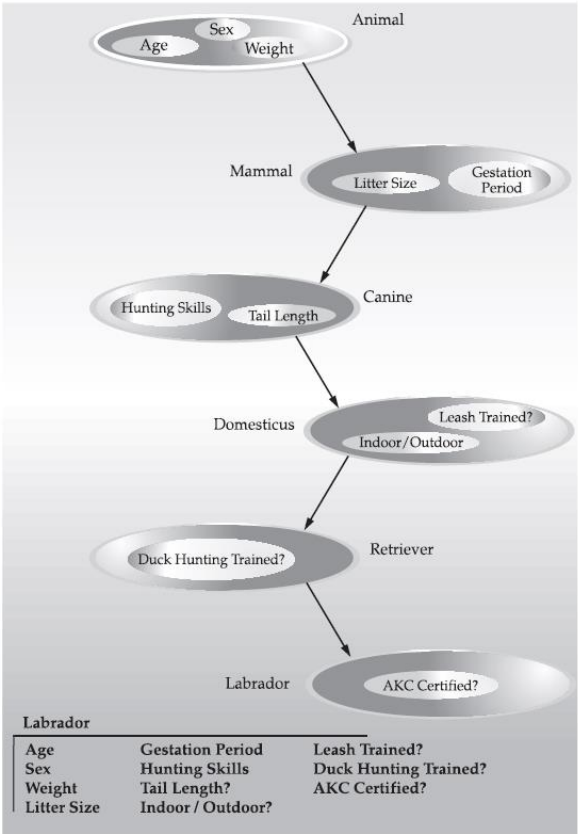


Fig 1. 5: Labrador inherits the encapsulation of all its super classes.

1.3. Java Buzz Words:

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Portable
4. Secured
5. Robust
6. Architecture neutral
7. Dynamic
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed

Simple:

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier.

Object-Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust:

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer will often 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 deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.)

Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to

read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can - and should - be managed by your program.

Multithreaded:

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multi-process synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behaviour of your program, not the multitasking subsystem.

Architecture Neutral:

A central issue for the Java designers was that of code longevity and portability. At the time of Java's creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance:

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed:

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic:

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

1.4. Overview of Java:

1.4.1 History of Java:

The history of java starts from Green Team. Java team members (also known as Green Team), initiated a revolutionary task to develop a language for digital devices such as

set-top boxes, televisions etc. For the green team members, it was an advance concept at that time. But it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

1) James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.
2) Originally designed for small, embedded systems in electronic appliances like set- top boxes.
3) Firstly, it was called "Greentalk" by James Gosling and file extension was ".gt".
4) After that, it was called Oak and was developed as a part of the Green project.

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 5.0 (30th Sep, 2004)
8. Java SE 6 (11th Dec, 2006)
9. Java SE 7 (28th July, 2011)
10. Java SE 8 (18th March, 2014)

1.4.2 Java Comments:

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments:

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

```
//This is single line comment
```

Example:

```
public class CommentExample1 {
    public static void main(String[] args) {
        int i=10;               //Here, i is a variable
        System.out.println(i);
    }
```

```
}
```

Output:

10

Java Multi Line Comment:

The multi-line comment is used to comment multiple lines of code.

Syntax:

```
/* This is
multi
line
comment
*/
```

Example:

```
public class CommentExample2 {
    public static void main(String[] args) {
        /* Let's declare and
        print variable in java. */
        int i=10;
        System.out.println(i);
    }
}
```

Output:

10

Java Documentation Comment:

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

```
/** This is
documentation comment
*/
```

Example:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
public static int sub(int a, int b){return a-b;}}
```

➔ **Compile** it by javac tool:
javac Calculator.java

➔ **Create Documentation API** by javadoc tool:
javadoc Calculator.java

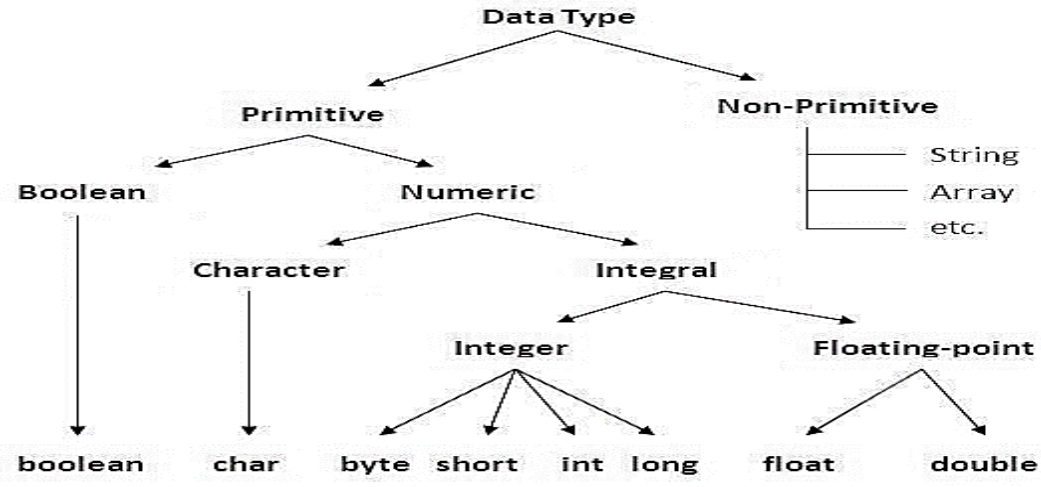
Name	Default Value	Width (in bits)	Range
boolean	false	1	true or false
char	'\u0000'	16	0 to 65535
long	0L	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	0	32	–2,147,483,648 to 2,147,483,647
short	0	16	–32,768 to 32,767
byte	0	8	–128 to 127
double	64	0.0d	4.9e–324 to 1.8e+308
float	32	0.0f	1.4e–045 to 3.4e+038

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

1.5. Data Types:

Java is a **Strongly Typed (Strongly Coupled)** Language. Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types




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

}

Output: 20

1.5.1 Integer Literals:

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. These are all decimal values, meaning they are describing a base 10 number. Two other bases that can be used in integer literals are octal (base eight) and hexadecimal (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal 0 to 7 range. Integer literals create an int value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as byte or long, without causing a type mismatch error.

You can also specify integer literals using binary. To do so, prefix the value with 0b or 0B. For example, this specifies the decimal value 10 using a binary literal:

int x = 0b1010;

You can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

int x = 123_456_789;

the value given **x** will be 123,456,789. The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid:

int x = 123___456___789;

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on. They are also useful for providing visual groupings when specifying binary literals. For example, binary values are often visually grouped in four-digits units, as shown here:

int x = 0b1101_0101_0001_1010;

1.5.2 Floating-Point Literals:

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either *standard* or *scientific* notation.

- ➔ *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers.
- ➔ *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.

Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can embed one or more underscores in a floating-point literal. This feature works the same as it does for integer literals,

which were just described. Its purpose is to make it easier to read large floating-point literals. When the literal is compiled, the underscores are discarded. For example, given

double num = 9_423_497_862.0;

the value given to num will be 9,423,497,862.0. The underscores will be ignored. It is also permissible to use underscores in the fractional portion of the number. For example,

double num = 9_423_497.1_0_9;

is legal. In this case, the fractional part is **.109**.

1.5.3 Boolean Literals:

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0.

1.5.4 Character Literals:

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'. The following table shows the character escape sequences:

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

1.5.5 String Literals:

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\n lines"
" \"This is in quotes\""

1.6. Variables and Arrays:

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. In addition, all variables have a scope, which defines their visibility, and a lifetime.

1.6.1 Declaring a Variable:

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

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

Example:

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;                    // declares three ints, a, b, and c.
int d = 3, e, f = 5;            // declares three more ints, initializing d and f.
byte z = 22;                    // initializes z.
double pi = 3.14159;            // declares an approximation of pi.
char x = 'x';                   // the variable x has the value 'x'.
```

1.6.2 Dynamic Initialization:

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

Example:

```
// Demonstrate dynamic initialization.
class DynInit {
    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);
    }
}
```

1.6.3 The Scope and Lifetime of Variables:

Java allows variables to be declared within any block as well as at the start of the main( ) method. A block is begun with an opening curly brace and ended by a closing curly brace. 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.

Many other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, object-oriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial. However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense.

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. Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested.

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

```
// Demonstrate block scope.
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);
    }
}
```

Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program:

```
// Demonstrate lifetime of a variable.
class LifeTime {
        public static void main(String args[]) {
            int x;
            for(x = 0; x < 3; x++) {
                int y = -1; // y is initialized each time block is entered
                System.out.println("y is: " + y); // this always prints -1
                y = 100;
                System.out.println("y is now: " + y);
            }
        }
}
```

1.6.4 Type Conversion and Casting:

To assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from double to byte. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions:

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. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

Casting Incompatible Types:

Although the automatic type conversions are helpful, they will not fulfil all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a 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

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
        public static void main(String args[]) {
                byte b;
                int i = 257;
                double d = 323.142;
                System.out.println("\nConversion of int to byte.");
                b = (byte) i;
                System.out.println("i and b " + i + " " + b);
                System.out.println("\nConversion of double to int.");
                i = (int) d;
                System.out.println("d and i " + d + " " + i);
                System.out.println("\nConversion of double to byte.");
                b = (byte) d;
                System.out.println("d and b " + d + " " + b);
        }
}
```

Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
class ByteAdd{
    public static void main(String[] args){
            byte a = 120, b = 5, c;
            c = (byte) (a + b);
            System.out.println(c);
    }}
```

1.7. Operators:

Java provides a rich operator environment. Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

1.7.1 Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
    public static void main(String args[]) {
            // arithmetic using integers
            System.out.println("Integer Arithmetic");
            int a = 1 + 1;
            int b = a * 3;
            int c = b / 4;
            int d = c - a;
            int e = -d;
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("c = " + c);
            System.out.println("d = " + d);
            System.out.println("e = " + e);
            // arithmetic using doubles
            System.out.println("\nFloating Point Arithmetic");
            double da = 1 + 1;
            double db = da * 3;
            double dc = db / 4;
            double dd = dc - a;
            double de = -dd;
            System.out.println("da = " + da);
            System.out.println("db = " + db);
            System.out.println("dc = " + dc);
            System.out.println("dd = " + dd);
            System.out.println("de = " + de);
```

```
    }
}
```

1.7.2 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:

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

```
// Demonstrate the bitwise logical operators.
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
        "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0 + 2 + 1 or 0011 in binary
        int b = 6; // 4 + 2 + 0 or 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b)|(a & ~b);
        int g = ~a & 0x0f;
        System.out.println(" a = " + binary[a]);
        System.out.println(" b = " + binary[b]);
        System.out.println(" a|b = " + binary[c]);
        System.out.println(" a&b = " + binary[d]);
        System.out.println(" a^b = " + binary[e]);
        System.out.println("~a&b|a&~b = " + binary[f]);
        System.out.println(" ~a = " + binary[g]);
    }
}
```

1.7.3 Relational Operators:

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

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

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.

1.7.4 Boolean Logical Operators:

The Boolean logical operators shown here operate only on boolean 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
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer. The logical ! operator inverts the Boolean state: !true == false and !false == true. The following table shows the effect of each logical operation:

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

Here is a program that is almost the same as the BitLogic example shown earlier, but it operates on boolean logical values instead of binary bits:

```
// Demonstrate the boolean logical operators.
class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
```

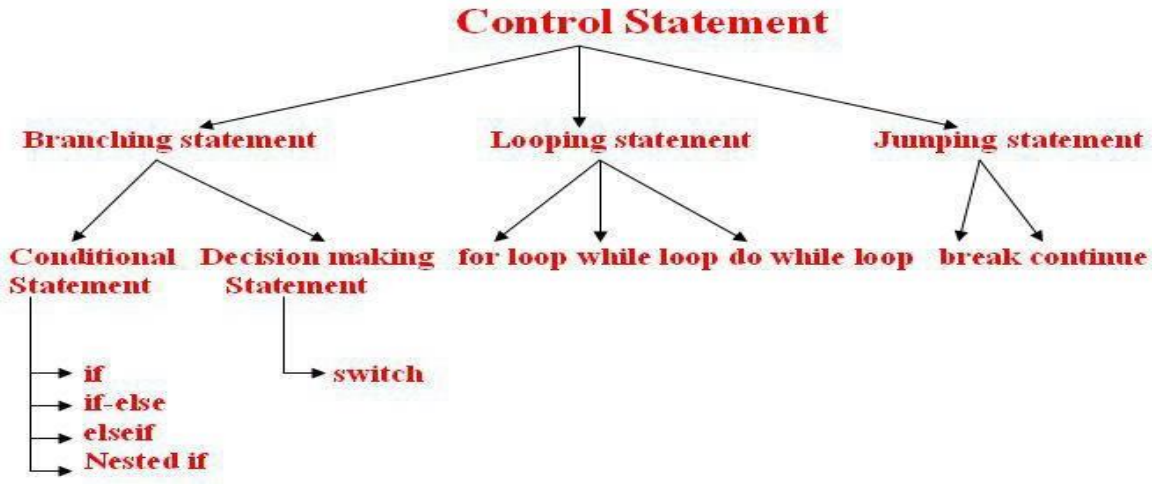
```
                boolean g = !a;
                System.out.println(" a = " + a);
                System.out.println(" b = " + b);
                System.out.println(" a|b = " + c);
                System.out.println(" a&b = " + d);
                System.out.println(" a^b = " + e);
                System.out.println("!a&b|a&!b = " + f);
                System.out.println(" !a = " + g);
        }
}
```

After running this program, you will see that the same logical rules apply to boolean values as they did to bits. As you can see from the following output, the string representation of a Java boolean value is one of the literal values true or false:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

1.8. Control Statements:

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion.



1.8.1 Java's Selection Statements:

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

a) if:

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;
```

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 if works like this: If the condition is true, then statement1 is executed.

if-else:

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

b) 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.

```
if (condition1) {
    if(condition2){
        statement1;
    }
}
```

c) The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder.

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

d) 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:

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
```

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

For versions of Java prior to JDK 7, *expression* must be of type **byte, short, int, char**, or an **enumeration**. Beginning with JDK 7, *expression* can also be of type **String**. Each value specified in the **case** statements must be a unique constant expression (such as a literal value). Duplicate **case** values are not allowed. The type of each value must be compatible with the type of expression. The **switch** statement works like this: The value of the expression is compared with each of the values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken. The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the **switch**.

1.8.2 Looping Statements:

Java's looping statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*.

a) 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) {
    // 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. The curly braces are unnecessary if only a single statement is being repeated.

```
// Demonstrate the while loop.
class While {
    public static void main(String args[]) {
        int n = 10;
        while(n > 0) {
            System.out.println("tick " + n);
            n--;
```

```
        }
    }
}
```

b) 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);
```

Ex:

```
// Demonstrate the do-while loop.
class DoWhile {
    public static void main(String args[]) {
        int n = 10;
        do {
            System.out.println("tick " + n);
            n--;
        } while(n > 0);
    }
}
```

c) for:

Beginning with JDK 5, 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. Here is the general form of the traditional for statement:

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

If only one statement is being repeated, there is no need for the curly braces. The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is executed only once. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

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, one at a time, from beginning to end. The collection being cycled through is specified by collection. There are various types of collections that can be used with the for, but the only type used in this chapter is the array. (Other types of collections that can be used with the for, such as those defined by the Collections Framework, are discussed later in this book.) With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.

Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, type must be compatible with the element type of the array.

For example, here is the preceding fragment rewritten using a for-each version of the for:

```
// Use a for-each style for loop.
class ForEach {
        public static void main(String args[]) {
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                int sum = 0;
                // use for-each style for to display and sum the values
                for(int x : nums) {
                        System.out.println("Value is: " + x);
                        sum += x;
                }
                System.out.println("Summation: " + sum);
        }
}
```

1.8.3 Jump Statements:

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program.

a) 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.

Using break to Exit a Loop:

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
// Using break to exit a loop.
class BreakLoop {
        public static void main(String args[]) {
                for(int i=0; i<100; i++) {
                if(i == 10) break; // terminate loop if i is 10
```

```
                    System.out.println("i: " + i);
                }
            System.out.println("Loop complete.");
            }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

Using break as a Form of Goto:

In addition to its uses with the switch statement and loops, the break statement can also be employed by itself to provide a "civilized" form of the goto statement. 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. It also prohibits certain compiler optimizations.

```
// Using break as a civilized form of goto.
class Break {
        public static void main(String args[]) {
                boolean t = true;
                first: {
                        second: {
                                third: {
                                System.out.println("Before the break.");
                                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.");
                }
        }
}
```

b) Using continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an

action. 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 the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses continue to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
        public static void main(String args[]) {
                for(int i=0; i<10; i++) {
                        System.out.print(i + " ");
                        if (i%2 == 0) continue;
                        System.out.println("");
                }
        }
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

c) Using 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.

```
// Demonstrate return.
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.");
        }
}
```

1.9. Arrays:

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

dataType[] arrayRefVar; // preferred way.

or

dataType arrayRefVar[]; // works but not preferred way.

Note: The style dataType[] arrayRefVar is preferred. The style dataType arrayRefVar[] comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:

double[] myList; // preferred way.

or

double myList[]; // works but not preferred way.

Creating Arrays:

You can create an array by using the new operator with the following syntax:

*arrayRefVar = **new** dataType[arraySize];*

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively, you can create arrays as follows:

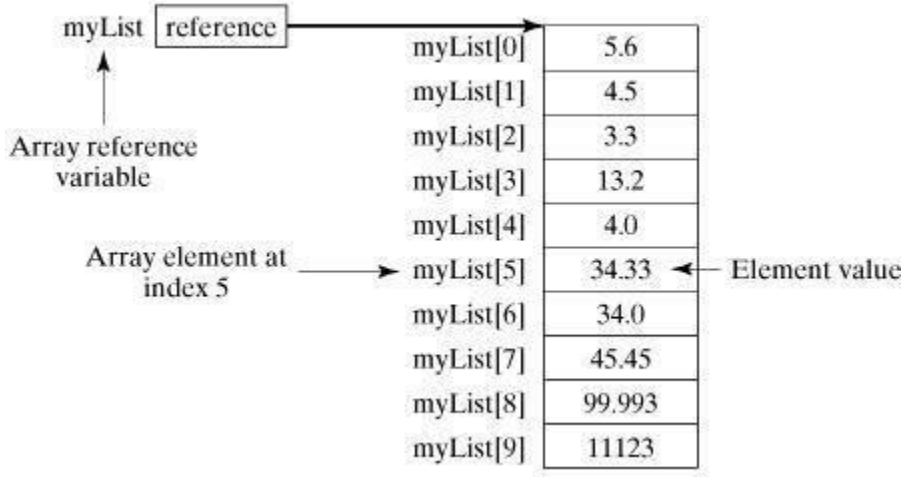
dataType[] arrayRefVar = {value0, value1, ..., valuek};

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

double[] myList = new double[10];

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

When processing array elements, we often use either for loop or for each loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {
        public static void main(String[] args) {
                double[] myList = {1.9, 2.9, 3.4, 3.5};
                // Print all the array elements
                for (int i = 0; i < myList.length; i++) {
                        System.out.println(myList[i] + " ");
                }
                // Summing all elements double total = 0;
                for (int i = 0; i < myList.length; i++) {
                        total += myList[i];
                }
                System.out.println("Total is " + total);
                // Finding the largest element
                double max = myList[0];
                for (int i = 1; i < myList.length; i++) {
                        if (myList[i] > max)
                                max = myList[i];
                        }
                System.out.println("Max is " + max);
        }
}
```

This would produce the following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

1.10. Introducing Classes:

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class.

1.10.1 The General Form of a Class:

When you define a class, you declare its exact form and nature. A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;
    type methodname1(parameter-list) {
            // body of method
    }
    type methodname2(parameter-list) {
            // body of method
    }
    // ...
    type methodnameN(parameter-list) {
            // body of method
    }
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public. Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a main( ) method at all.

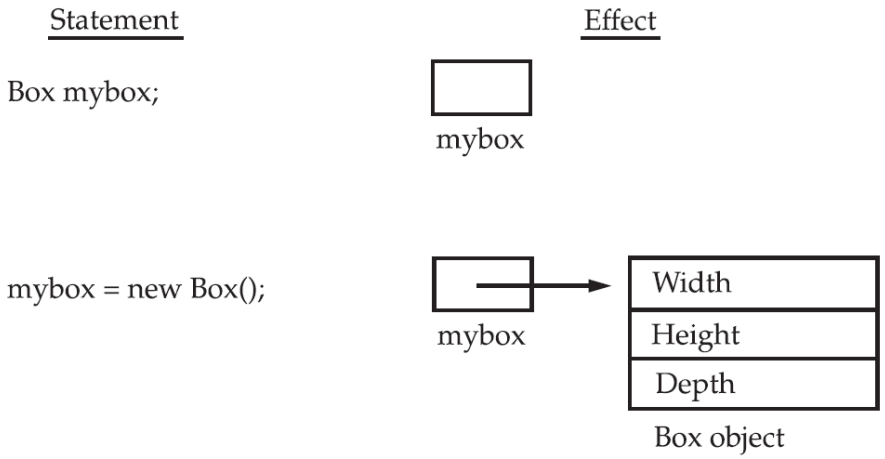
1.10.2 A Simple Class:

Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods (but some will be added soon).

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

As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```



Example:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
        double width;
        double height;
        double depth;
}
// This class declares an object of type Box.
class BoxDemo {
        public static void main(String args[]) {
                Box mybox = new Box();
                double vol;
                // assign values to mybox's instance variables
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                // compute volume of box
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

1.10.3 Assigning Object Reference Variables:

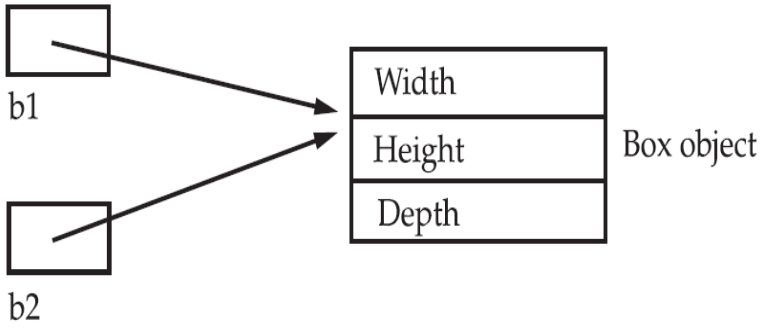
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes

b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is described here in the following picture:



1.10.4 Introducing Methods:

Classes usually consist of two things: instance variables and methods. This is the general form of a method:

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

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty. Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

return value;

Here, value is the value returned.

1.10.5 Adding a Method to a Class:

Let's begin by adding a method to the Box class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the Box class rather than the BoxDemo class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the Box class compute it. To do this, you must add a method to Box, as shown here:

```
// This program includes a method inside the box class.
class Box {
    double width;
    double height;
    double depth;
    // display volume of a box
    void volume() {
            System.out.print("Volume is ");
```

```
            System.out.println(width * height * depth);
        }
    }
    class BoxDemo {
        public static void main(String args[]) {
            Box mybox1 = new Box();
            Box mybox2 = new Box();
            // assign values to mybox1's instance variables
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            /* assign different values to mybox2's
            instance variables */
            mybox2.width = 3;
            mybox2.height = 6;
            mybox2.depth = 9;
            // display volume of first box
            mybox1.volume();
            // display volume of second box
            mybox2.volume();
        }
    }
```

1.10.6 Constructors:

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Constructor in java is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors:

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default Constructor:

A constructor that have no parameter is known as default constructor.

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double w, h, d;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
W = h = d = 10;
}
// compute and return volume
double volume() {
return w * h * d;
}
}
class BoxDemo {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

Parameterized Constructors:

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
```

```
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

1.10.7 The "this" Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the "this" keyword. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

To better understand what this refers to, consider the following version of Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
        this.w = w;
        this.h = h;
        this.d = d;
}
```

This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.

1.10.8 Garbage Collection:

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

1.11.Methods and Classes:

1.11.1 Overloading Methods:

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class OverloadDemo {
    void test() {
            System.out.println("No parameters");
    }
    // Overload test for one integer parameter.
    void test(int a) {
            System.out.println("a: " + a);
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
            System.out.println("a and b: " + a + " " + b);
    }
    // Overload test for a double parameter
    double test(double a) {
            System.out.println("double a: " + a);
            return a*a;
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // Overload test for a double parameter
    void test(double a) {
        System.out.println("Inside test(double) a: " + a);
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;
        ob.test();
        ob.test(10, 20);
        ob.test(i); // this will invoke test(double)
        ob.test(123.2); // this will invoke test(double)
    }
}
```

1.11.2 Overloading Constructors:

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the Box class developed in the preceding chapter. Following is the latest version of Box:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
```

```
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

1.11.3 Using Objects as Parameters:

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equalTo(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

1.11.4 Access Control:

Access of the data in Java can be controlled by the Access Modifiers. As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

1. **Default:** When no access modifier is specified for a class, method or data member – It is said to be having the **default** access modifier by default. The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.
2. Private: The private access modifier is specified using the keyword **private**.
    - The methods or data members declared as private are accessible only **within the class** in which they are declared.

- Any other **class of same package will not be able to access** these members.
- Top level Classes or interface cannot be declared as private because

i. private means "only visible within the enclosing class".

ii. protected means "only visible within the enclosing class and any subclasses"

Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

3. **protected:** The protected access modifier is specified using the keyword **protected**.
    - The methods or data members declared as protected are **accessible within same package or sub classes in different package.**
4. **public:** The public access modifier is specified using the keyword **public**.
    - The public access modifier has the **widest scope** among all other access modifiers.
    - Classes, methods or data members which are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of a public data members.

1.11.5 The static Keyword:

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). The main( ) is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
```

```
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

Here is the output of the program:

Static block initialized.
x = 42
a = 3
b = 12

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a static method from outside its class, you can do so using the following general form:

classname.method( )

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

1.11.6 The final Keyword:

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

1.11.7 Nested and Inner Classes:

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example:

```
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;
    void test() {
            Inner inner = new Inner();
            inner.display();
    }
    // this is an inner class
    class Inner {
            void display() {
            System.out.println("display: outer_x = " + outer_x);
            }
    }
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Example:

```
class Outer{
    private int data=30;
    class Inner{
            void msg(){System.out.println("data is "+data);}
    }
    public static void main(String args[]){
```

```
        Outer obj=new Outer();
        Outer.Inner in=obj.new Inner();
        in.msg();
    }
}
```

Anonymous Inner Class:

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Example:

```
abstract class Eatable{
static int data=30;
abstract void eat();
}
class AnonymousInner{
public static void main(String args[]){
Eatable obj=new Eatable(){
  void eat(){System.out.println("data is "+data);}
};
obj.eat();
}
}
```

Java anonymous inner class example using interface:

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Static Nested Class:

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Example:

```
class TestOuter1{
  static int data=30;
```

```
 static class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
}
```

1.12.String Handling:

String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type String. Even string constants are actually String objects. For example, in the statement:

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a String object.

The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String**, called **StringBuffer** and **StringBuilder**, which allow strings to be altered, so all of the normal string manipulations are still available in Java.

The following program demonstrates the ways of the constructing Strings:

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

1.12.1 Methods of String Class:

The String class contains several methods that you can use. Here are a few. You can test two strings for equality by using equals( ). You can obtain the length of a string by calling the length( ) method. You can obtain the character at a specified index within a string by calling charAt( ). The general forms of these three methods are shown here:

boolean equals(secondStr)
boolean equalsIgnoreCase(secondStr)
int length()
char charAt(index)

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = “first String”;
System.out.println("Length of strOb1: " +
strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
if(strOb1.equalsIgnoreCase(strOb2))
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equalsIgnoreCase(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
}
```

1.12.2 Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ). A command-line argument is the information that directly follows the program’s name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a String array passed to the args parameter of main( ). The first command-line argument is stored at args[0], the second at args[1], and so on. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

1.12.3 Varargs: Variable-Length Arguments:

Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short for variable-length arguments. A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
static void vaTest(int v[]) {
System.out.print("Number of args: " + v.length +
" Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how an array must be created to
// hold the arguments.
int n1[] = { 10 };
int n2[] = { 1, 2, 3 };
int n3[] = { };
vaTest(n1); // 1 arg
vaTest(n2); // 3 args
vaTest(n3); // no args
}
}
```

In the program, the method vaTest( ) is passed its arguments through the array v. This old-style approach to variable-length arguments does enable vaTest( ) to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling vaTest( ). Not only is it tedious to construct an array each time vaTest( ) is called, it is potentially error-prone. The varargs feature offers a simpler, better option. A variable-length argument is specified by three periods (…). For example, here is how vaTest( ) is written using a vararg:

```
static void vaTest(int ... v) {
```

```
// Demonstrate variable-length arguments.
class VarArgs {
// vaTest() now uses a vararg.
static void vaTest(int ... v) {
System.out.print("Number of args: " + v.length +
" Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how vaTest() can be called with a
// variable number of arguments.
vaTest(10); // 1 arg
vaTest(1, 2, 3); // 3 args
vaTest(); // no args
}
}
```