Unit 1

Introduction

Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects. Computer programs, such as the Java programs you'll read in this book interacting software objects.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are ―alive‖ they in move some around and sense do things. ―Inanimate objects, on the other hand, do not move on their own .Objects of both types, however, have some things in common. They all have attributes (e.g., size, shape, colour and weight), and they all exhibit behaviour's (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleep crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We will study the kinds of attributes and behaviour's that software objects have. Humans learn about existing objects by studying their attributes and observing their behaviour's. Different objects can have similar attributes and can exhibit similar behaviour's. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Object-oriented design provides a natural and intuitive way to view the software design process—namely, modelling objects by their attributes and behaviour's just as we describe real- world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

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 of my way model. The object model in Java such as integers, are kept as high-performance non objects.

OOD encapsulates (i.e., wraps) attributes and operations (behaviour's) into objects, an object's attributes and operations are information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented ,implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the wheel and so on. Information hiding, as we will see, is crucial to good software engineering.

Languages like Java are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement an object-oriented design as a working system. Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. In Java, the unit of programming is the class from which objects are eventually instantiated (created). Java classes contain methods (which implement operations and are similar to functions in C) as well as fields (which implement attributes).

Java programmers concentrate on creating classes. Each class contains fields, and the set of methods that manipulate the fields and provide services to clients (i.e., other classes that use the class). The programmer uses existing classes as the building blocks for constructing new classes. Classes are to objects as blueprints are to houses. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the ―bank teller‖ class needs to relate ―safe‖ class, and so on.

Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components. Just as realtors often say that the three most important factors affecting location, people in the software community affecting the future of software development are ―reuse, classes when building new classes and programs saves time and effort. Reuse also helps programmers build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.

Indeed, with object technology, you can build much of the software you will need by combining classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software development efforts.

History of Java

- Java started out as a research project.
- Research began in 1991 as the Green Project at Sun Microsystems, Inc.
- Research efforts birthed a new language, OAK. (A tree outside of the window of James Gosling's office at Sun).
- It was developed as an embedded programming language, which would enable embedded system application.
- It was not really created as web programming language.
- Java is available as jdk and it is an open source s/w.
- Language was created with 5 main goals:
    - It should be object oriented.
    - A single representation of a program could be executed on multiple operating systems. (i.e. write once, run anywhere)
    - It should fully support network programming.
    - It should execute code from remote sources securely.
    - It should be easy to use.
- Oak was renamed Java in 1994.
- Now Sun Microsystems is a subsidiary of Oracle Corporation.

Java Platforms

There are three main platforms for Java:

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

Java Terminology

Java Development Kit:

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

Java Virtual Machine:

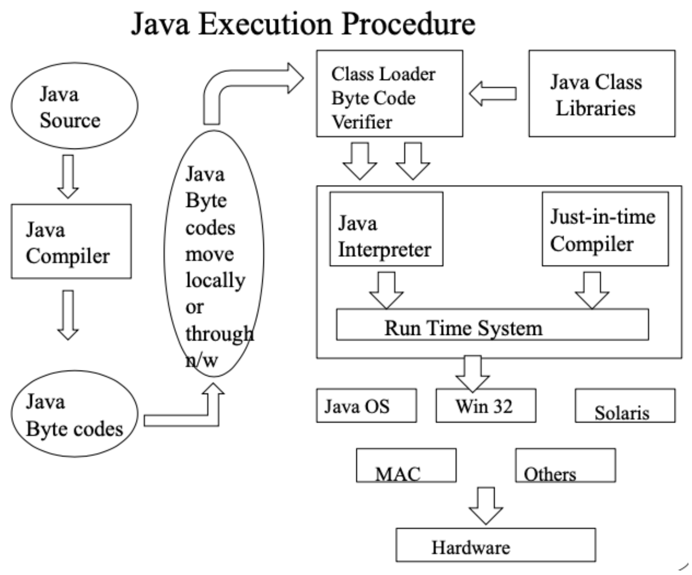
An abstract machine architecture specified by the Java Virtual Machine Specification.

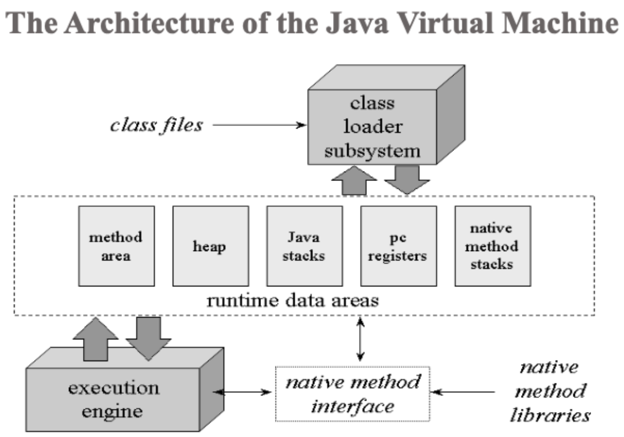
It interprets the byte code into the machine code depending upon the underlying OS and hardware combination. JVM is platform dependent. (It uses the class libraries, and other supporting files provided in JRE)

Java Runtime Environment:

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

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





Java Virtual Machine

- Class loader subsystem: A mechanism for loading types (classes and interfaces) given fully qualified names.
- The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.
- Each Java virtual machine also has an execution engine: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

Class loader subsystem

- The Java virtual machine contains two kinds of class loaders: a bootstrap class loader and user-defined class loaders.
- The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application.
- Loading: finding and importing the binary data for a type
- Linking: performing verification, preparation, and (optionally) resolution
- Verification: ensuring the correctness of the imported type
- Preparation: allocating memory for class variables and initializing the memory to default values
- Resolution: transforming symbolic references from the type into direct references.
- Initialization: invoking Java code that initializes class variables to their proper starting values.
- When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file.
- It places this type information into the method area.
- As the program runs, the virtual machine places all objects the program instantiates onto the heap.
- As each new thread comes into existence, it gets its own pc register (program counter) and Java stack.
- Byte code is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).The JVM is an interpreter for byte code.

Object Oriented Programming Concepts

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

Data 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 often of thousands of individual parts. They think of it as a well-defined object with its own unique behaviour. 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.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player.

The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

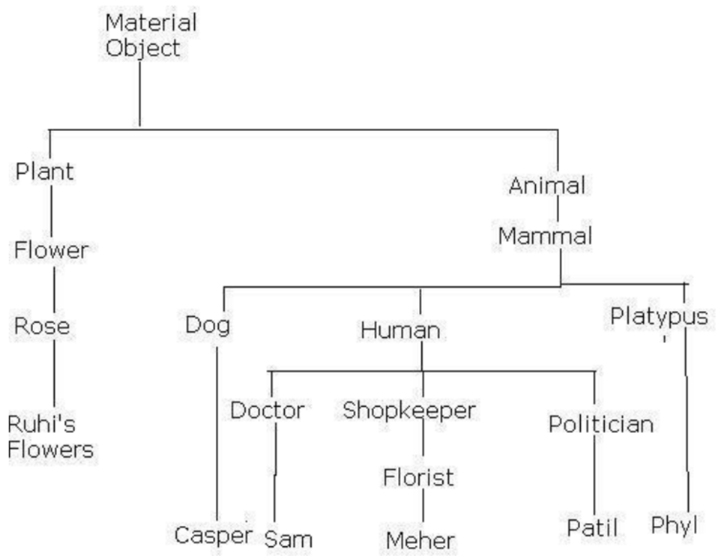
Encapsulation

An object encapsulates the methods and data that are contained inside it .the rest of the system interacts with an object only through a well-defined set of services that it provides.

Inheritance

I have more information about Flora –not necessarily because she is a florist but because she is a shopkeeper.

One way to think about how I have organized my knowledge of Flora is in terms of a hierarchy of categories:


Material Object
Plant
Flower
Rose
Ruhi's Flowers
Animal
Mammal
Dog
Human
Platypus
Doctor
Shopkeeper
Politician
Florist
Casper
Sam
Meher
Patil
Phyl

Classes and Objects

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapters primarily exist simply to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax.

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, you will often see the two words object and instance used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.

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. The 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, it is the methods that de can be used.

Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new.

Ex: Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object mybox = new Box(); // allocate a Box object

A Closer Look at new

As just explained, the new operator dynamically allocates memory for an object. It has this general form:

class-var = new classname( );

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box.

Difference between Procedural Programming and Object oriented Programming

Procedural	OO programming
Code is placed into totally distinct functions or procedures	Everything treated as an Object
Data placed in separate structures and is manipulated by these functions or procedures	Every object consist of attributes(data) and behaviors (methods)
Code maintenance and reuse is difficult	Code maintenance and reuse is easy
Data is uncontrolled and unpredictable (i.e. multiple functions may have access to the global data)	The data of an object can be accessed only by the methods associated with the object
You have no control over who has access to the data	Good control over data access
Testing and debugging are much more difficult	Testing and debugging are much easy
Not easy to upgrade	Easy to upgrade
Not easy to partition the work in a project	Easy to partition the work in a project

Features of JAVA (Java Buzz Words)

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

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. 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. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.. Object-Oriented Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the everything is an object paradigm and the programming.

Robust

The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.

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 ask in traditional programming environments. For example, in C/C++, the pro grammer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you.

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 multiprocess .synchronization that enables you to construct smoothly running interactive systems.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system up grades, 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 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 interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs.

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. The original version of Java (Oak) included features for intra address-space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/server programming.

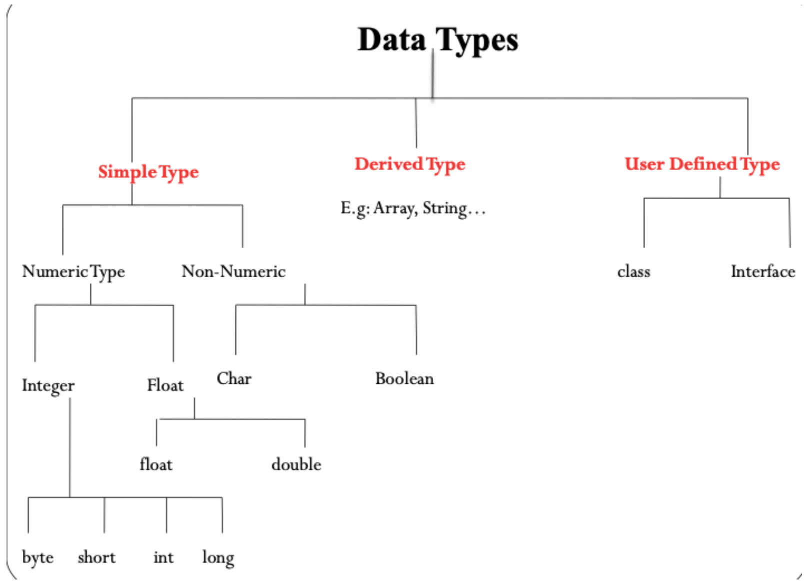
Dynamic

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 applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

Keywords

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

DATA TYPES



Java defines eight simple (or elemental) types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups:

Integers This group includes byte, short, int, and long, which are for whole valued signed numbers. Floating-point numbers This group includes float and double, which represent numbers with fractional precision. Characters This group includes char, which represents symbols in a character set, like letters and numbers. Boolean This group includes boolean, which is a special type for representing true/false values.

Integers

Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other Computer languages, including C/C++, support both signed and unsigned integers.

Name

long int short

byte

Width

64 32

16 8

Range

–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 –2,147,483,648 to 2,147,483,647

–32,768 to 32,767 –128 to 127

byte
The smallest integer type is byte. This is a signed 8-bit type that has a range from –128to

127. Variables of type byte are especially useful when you're a network or file. They are also useful not be when directly compatible-intypes with. Java's other built

Syntax: byte b, c;

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called big-endian format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.

Here are some examples of short variable declarations: short s;

short t; int

The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. Any time you have an integer expression involving bytes, shorts, ints, and literal numbers, the entire expression Is promoted to int before the calculation is done.

long

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

Floating-Point
Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.

Their width and ranges are shown here:

Name Width Bits Approximate Range double 64 4.9e–324 to 1.8e+308

float 32 float

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require example, float can be useful when representing dollars and cents.

Here are some example float variable declarations: float hightemp, lowtemp; double

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

Here is a short program that uses double variables to compute the area of a circle: // Compute the area of a circle.

```
class Area {

public static void main(String args[]) {

double pi, r, a;

r = 10.8;

// radius of circle pi = 3.1416;

// pi, approximately

a = pi * r * r;

// compute area

System.out.println("Area of circle is " + a);

}
}
```

Characters

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is an integer type that is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characters.. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1,ranges from 0 to 255. Booleans

Java has a simple type, called Boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, such as a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

Here is a program that demonstrates the boolean type:

There are three interesting things to notice about this program. First, as you can see, when a Boolean value is output by println( ), ―true‖ or ―false‖ Second,the value of a boolea variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this:

```
if(b == true)
...
```

Third, the outcome of a relational operator, such as <, is a boolean value. This is why the expression 10 > 9 displays the value ―true.‖ Further, around 10 > 9 is necessary because the + operator has a higher precedence than the >.

Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer. Declaring a Variable
- In Java, all variables must be declared before they can be used. type identifier [= value][, identifier [= value] ...] ;

Types of variables

- Instance Variable
- Class Variable
- Local Variable
- Parameters

Local variables :

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

Instance variables :

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

Class/Static variables :

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.

- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are stored in static memory.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables.
- Default values are same as instance variables.
- Static variables can be accessed by calling with the class name Class Name. Variable Name

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] ...] ;

The identifier is the name of the variable.

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

int a, b, c;

int d = 3, e, f = 5;

// declares three ints, a, b, and c.

// declares three more ints, initializing

// declares an approximation of pi. // the variable x has the value 'x'.

byte z = 22;

double pi = 3.14159;

char x = 'x';

The Scope and Lifetime of Variables

Scope

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

Lifetime

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

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

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

Variables declared inside a scope are not accessible to code outside.

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

```
public class Scope {

public static void main(String args[]){

int x; //know to all code within main x=10;

if(x==10){ // starts new scope

int y=20; //Known only to this block

//x and y both known here System.out.println("x and y: "+x+" "+y); x=y+2;

}

// y=100; // error ! y not known here //x is still known here System.out.println("x is "+x);

} }
```

Arrays

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

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

type var-name[ ];

Here, type declares the base type of the array. The base type determines the data

type of each element that comprises the array.

// Demonstrate a one-dimensional array. class Array {

public static void main(String args[]) { int month_days[];

month_days = new int[12];

month_days[0] = 31;

month_days[1] = 28;

month_days[2] = 31;

month_days[3] = 30;

month_days[4] = 31;

month_days[5] = 30;

month_days[6] = 31;

month_days[7] = 31;

month_days[8] = 30;

month_days[9] = 31;

month_days[10] = 30;

month_days[11] = 31;

System.out.println("April has " + month_days[3] + " days."); } }

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you mightexpect, look and act like regular multidimensional arrays. However, as you will see there are a couple of subtle differences. To declare a multidimensional array variable,specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

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

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an

array of arrays of int.

```
// Demonstrate a two-dimensional

array. class TwoDArray {

public static void main(String args[]) { int twoD[][]= new int[4][5];

int i, j, k = 0; for(i=0; i<4; i++) for(j=0; j<5; j++) { twoD[i][j] = k; k++;

}

for(i=0; i<4; i++) { for(j=0; j<5; j++)

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

System.out.println();

}
}

}
```

This program generates the following output: 0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two dimensional array in which the sizes of the second dimension are unequal.

Operators

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

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

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

```
class IncDec{

public static void main(String args[]){

int a = 1; int b = 2; int c,d;

c = ++b;

d = a++;

c++;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

} }
```

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

The Bitwise Operators

Java defines several bitwise operators which 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

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

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.

```
public class RelationalOperatorsDemo {

public static void main(String args[]) {

int x=10,y=5; System.out.println(“x>y:”+(x>y));

System.out.println(“x<y:”+(x<y));

System.out.println(“x>=y:”+(x>=y));
```

```
System.out.println("x<=y:"+(x<=y));

System.out.println("x==y:"+(x==y));

System.out.println("x!=y:"+(x!=y));

} }
```

Boolean Logical Operators

- The Boolean logical operators 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

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

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

```
class Test{

public static void main(String args[]){

} }

int denom=0,num=20;

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

System.out.println("Hi");

public class TernaryOperatorDemo {

public static void main(String args[]) {

int x=10,y=12;

int z;
```

```
    z= x > y ? x : y; System.out.println("Z="+z);
}
```

The Assignment Operator

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form: var = expression;

Here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment: int x, y, z;

x = y = z = 100;

// set x, y,

and z to 100

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a ―cha assignment‖ is an easy way to set a group of

The ? Operator

Java includes a special ternary (three-way) operator that can replace certain types of if- then-else statements. This operator is the ?, and it works in Java much like it does in C, C++, and C#. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a Boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same void type,. which can't be

Control statements

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

It is examined in detail here. The if statement is Java' conditional program branch execution state through two different paths. Here is the general form of the if statement:

if (condition) statement1;

else statement2;

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

else clause is optional.

int a,

b; // ...

if(a < b) a = 0;

else b = 0;

The if-else-if Ladder

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

ladder. It looks like this:

if(condition)

statement;
else if(condition)

statement;

else if(condition) statement;

...

else

statement;

switch

The switch statement is Java's multiway branch 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 }

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed

Iteration Statements

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

While

The while loop is Java's fundamental looping most 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.

do-while

As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a while loop at least once, even if the conditional expression is false to begin with.

Syntax:

```
do{

// body of loop

} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
// 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);

}
}
```

For

As you will see, it is a powerful and versatile construct. Here is the general form of the 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.. 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.

```
// Demonstrate the for loop. class ForTick {

public static void main(String args[]) {

int n;

for(n=10; n>0; n--)

System.out.println("tick " + n);

 }
}
```

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.

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. Although a full discussion of return must wait until methods are discussed in Chapter 7, a brief look at return is presented here.

As you can see, the final println( ) statement is not executed. As soon as return is executed, control passes back to the caller.

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common 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.

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 sou

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.

It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

int a;

byte b;

// ...

b = (byte) a;

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be

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;
be
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);
}
}
```

This program generates the following output:

Conversion of int to byte. i and b 257 1

Conversion of double to int. d and i 323.142 323 Conversion of double to byte. d and b 323.142 67

SIMPLE JAVA PROGRAM

```
/*

This is a simple Java program. Call this file "Example.java". */

class Example {

// Your program begins with a call to main().

public static void main(String args[]) {

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

}

}
```

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

static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main ( ) is called by the Java interpreter before any objects are made.

void states that the main method will not return any value. main() is called when a Java application begins. In order to run

a class, the class must have a main method.

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

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

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

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

Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control.

How a member can be accessed is determined by the access specifier that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will public, private be, and easy protected. . Java's Java also defines a default access level. protected applies only when inheritance is involved. The other access specifiers are described next. Let's begin public by and private defining. When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. Now you can understand why main( ) has always been preceded by the public specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

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. You can use this

anywhere a reference to an object of the cur what this refers to, consider the following version of Box( ):

// A redundant use of this. Box(double w, double h, double d)

{ this.width = w;

this.height = h;

this.depth = 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. While it is redundant in this case, this is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which instance variables. However, when a local variable has the same variable has the name as an instance variable, the local variable hides the instance variable.

```
// Use this to resolve name-space collisions.

Box(double width, double height, double depth)

{

 this.width = width;

this.height = height;

this.depth = depth;

}
```

A word of caution: The use of this in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

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. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

Overloading methods and constructors

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 implements polymorphism.

```
// 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
```

As you can see, test( ) is overloaded four times.

Overloading Constructor

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 Box understand class why developed in the preceding chapter. Following is the latest version of 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; }
}
```

Argument/Parameter passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This method copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed.

For example, consider the following

program: // Simple types are passed by value.

```
class Test {

void meth(int i, int j)

{ i *= 2;

j /= 2;

}

}

class CallByValue {

public static void main(String args[]) {

Test ob = new Test();

int a = 15, b = 20;

System.out.println("a and b before call: "+ a + " " + b);

ob.meth(a, b);

System.out.println("a and b after call: " +a + " " + b);

}
}
```

The output from this program is shown here: a and b before call: 15 20 a and b after call: 15 20

Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

```
// A simple example of recursion(factorial). class Factorial {

// this is a recursive function

int fact(int n) {

int result;

if(n==1) return 1; result = fact(n-1) * n; return result;

} }

class Recursion {

public static void main(String args[]) {

Factorial f = new Factorial();

System.out.println("Factorial of 3 is " + f.fact(3));

System.out.println("Factorial of 4 is " + f.fact(4));

System.out.println("Factorial of 5 is " + f.fact(5));

}

}
```

The output from this program is shown here: Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

String class

Although the String class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. String is probably the most commonly used class in Java's class library. The obvious area very important reason part of for 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 String is constant. Fortunately ,String, Java handles too‖ String is constants in the same way that other computer language worry about this.

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 the can modifications.
- Java defines String ,called peer String Buffer class, which allows of strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

String myString = "this is a test";

Once you have created a String object, you can use it anywhere that a string is allowed. For

example, this statement displays myString:

System.out.println(myString);

Java defines one operator for String objects: +. It is used to concatenate two strings.

For example, this statement

String myString = "I" + " like " + "Java."; results in myString containing ―I like Java.‖

The following program demonstrates the preceding concepts: // 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);

}
}

The output produced by this program is shown here: First String

Second String

First String and Second String

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(String object)

int length( )

char charAt(int 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 = strOb1;

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"); }

}

This program generates the following output: Length of strOb1: 12

Char at index 3 in strOb1: s strOb1 != strOb2

strOb1 == strOb3

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays. class StringDemo3 {

public static void main(String args[]) { String str[] = { "one", "two", "three" }; for(int i=0;

i<str.length; i++) System.out.println("str[" + i + "]: " + str[i]); }

}
```

Here is the output from this program: str[0]: one

str[1]: two str[2]: three

As you will see in the following section, string arrays play an important part in many Java programs.