



Java Programming


Unit – 1

Introduction to OOP





TABLE OF CONTENTS

- 01 Introduction - Need of OOP
 - 02 Principles of OOP
 3. Java Features
 4. Installation of JDK
- 



Object Oriented Programming





Object Oriented Programming

- Object oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area of both data and functions that can be used as templates for creating copies of such modules on demand.
- Object is considered to be a partitioned area of computer memory that stores data and set of operations that can access the data.



Introduction and Need of OOP

- Object-Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour.
- The object-oriented programming approach encourages:
 1. Modularisation: where the application can be decomposed into modules.
 2. Software re-use: where an application can be composed from existing and new modules.



Principles of Object Oriented Programming

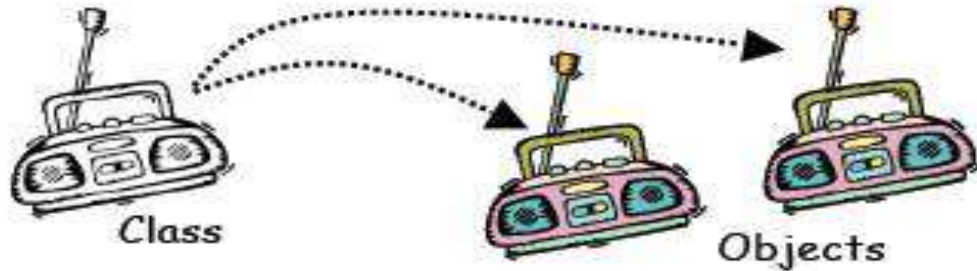
- **Object** means a real world entity such as pen, chair, table etc.
- **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. **Object**
2. **Class**
3. **Encapsulation**
4. **Inheritance**
5. **Polymorphism**
6. **Abstraction**



Class

- A class is a set of objects with:
 - Same interface, same behavior, **different identify**
- Classes allow modeling of simple classification in the real world, and simplify design
- An object is an “instance” of a class





Objects

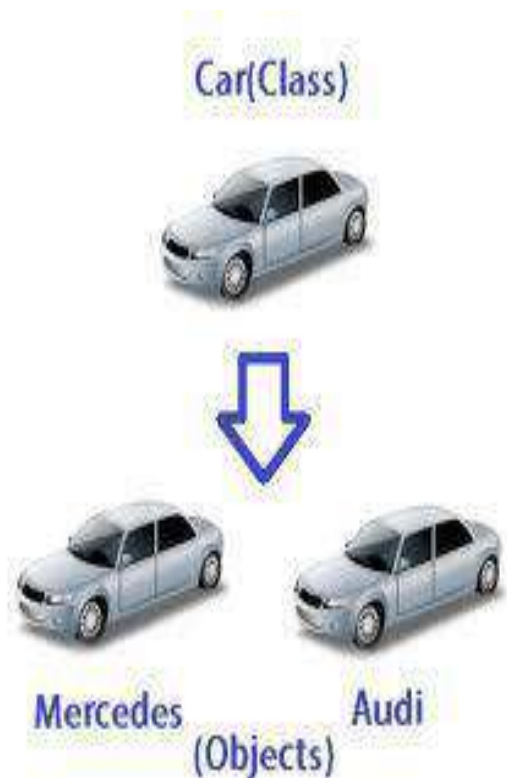
- An object has “memory”

Object State:

- **A bank account:** a number, a name, an address, a balance, an overdraft limit, a branch, a PIN number

Object Behaviour

- **A bank account:** withdraw, deposit, calculate interest, print statement.





Encapsulation

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

- **Data Fields are private**
- **Constructors and accessor methods are defined**

Person
-name : String
-age : int
+Person(String name, int age)
+getName() : String
+setName(String name)
+getAge() : int

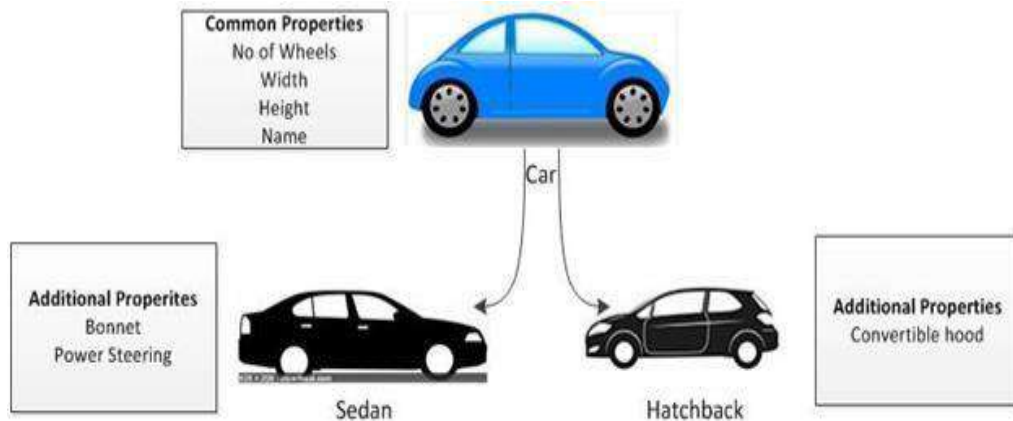


Capsule



Inheritance

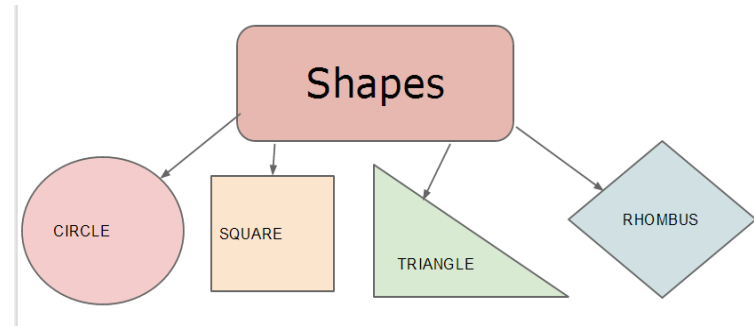
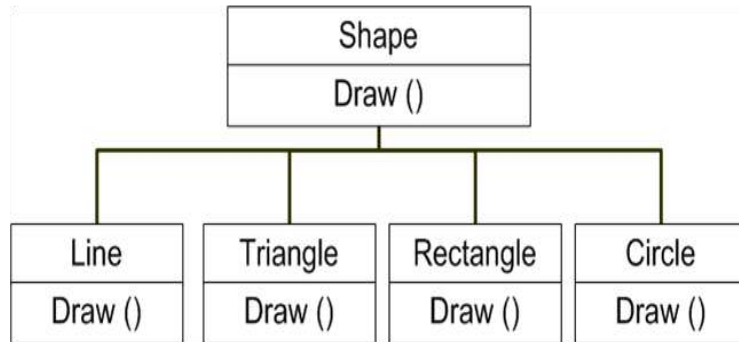
- When one object acquires all the properties and behaviours of parent object i.e. known as **Inheritance**.
- It provides code reusability.





Polymorphism

- A property of object oriented software by which an abstract operation may be performed in different ways in different classes.
 - Requires that there be multiple methods of the same name
 - The choice of which one to execute depends on the object.





Abstraction

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





In Real Life...



Abstraction



Login Validation
Process Hidden

Encapsulation



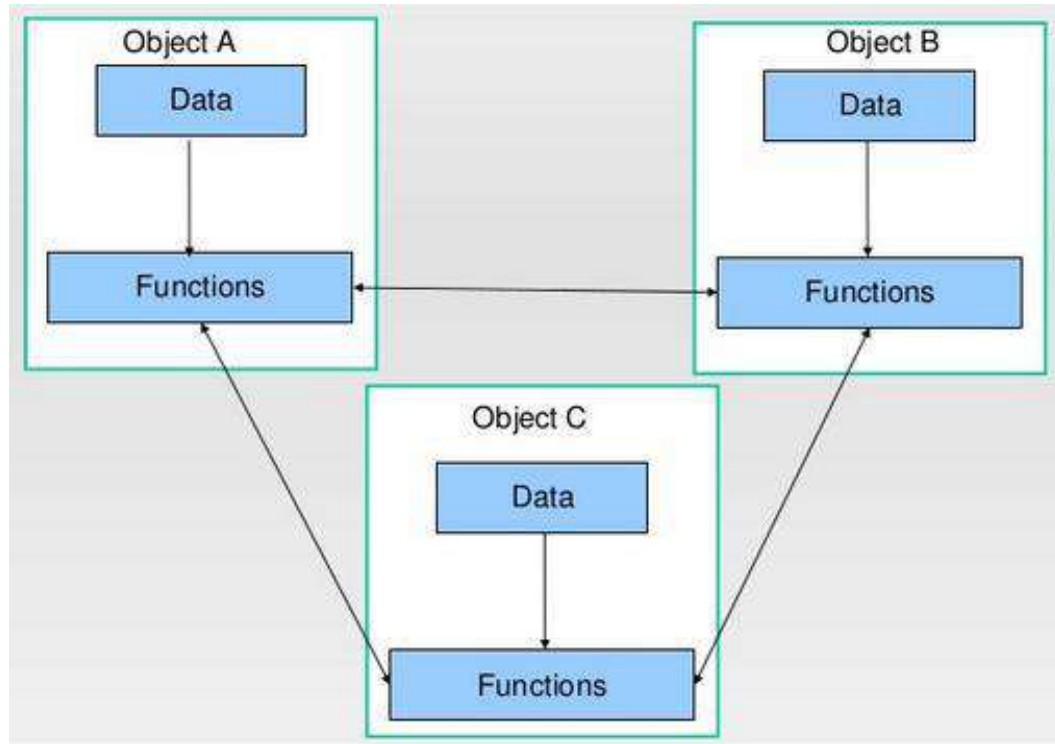
Inheritance



Polymorphism



Organization of data & function OOP





Striking features of OOP

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects
- Functions that operate on the data of an object are tied together in the data structure
- Data is hidden and cannot be accessed by external functions
- Objects may communicate with each other through functions
- New data and functions can be easily added whenever necessary.



Procedural vs. Object-Oriented Programming

- The unit in procedural programming is *function*, and unit in object-oriented programming is *class*
- Procedural programming concentrates on creating functions, while object-oriented programming starts from isolating the classes, and then look for the methods inside them.
- Procedural programming separates the data of the program from the operations that manipulate the data, while object-oriented programming focus on both of them



Applications of OOP

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.



Advantages of OOP

- Code reuse & recycling
- Improved software-development productivity
- Improved software maintainability
- Faster development
- Lower cost of development
- Higher-quality software
- Encapsulation



History of Java

- Java is related to C++, which is a direct descendant of C.
- From C, Java derives its syntax.
- Many of Java's object-oriented features were influenced by C++.
- HotJava
 - The first Java-enabled Web browser
- JDK Evolutions
- J2SE, J2ME, and J2EE



The Creation of Java

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995.
- Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments.
- The second force was the World Wide Web.



JDK

Versions

- JDK 1.02 (1995)
- JDK 1.1 (1996)
- Java 2 SDK v 1.2 (JDK 1.2, 1998)
- Java 2 SDK v 1.3 (JDK 1.3, 2000)
- Java 2 SDK v 1.4 (JDK 1.4, 2002)
- J2SE 5 (JDK 1.5, 2008)
- J2SE 6 (2010)



JDK Editions

- Java Standard Edition (J2SE)
 - J2SE can be used to develop client-side standalone applications or applets.
- Java Enterprise Edition (J2EE)
 - J2EE can be used to develop server-side applications such as Java servlets and Java Server Pages.
- Java Micro Edition (J2ME).
 - J2ME can be used to develop applications for mobile devices such as cell phones.



Java Compiled & Interpreted

- Usually a computer language is either uses compiler or interpreter.
- But Java Combines both these approaches.
- First Java Compiler translates source code into byte code instructions.
Byte code is not machine instructions.
- Second Java Interpreter directly executes the byte code.



Compile-time Environment

Java Source
(.java)

Java Compiler

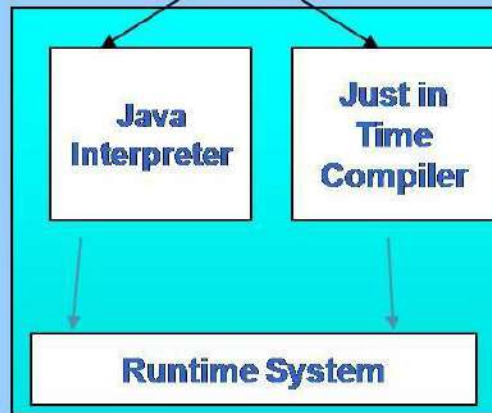
Java Bytecode
(.class)

Java Bytecodes
move locally
or through
network

Compile-time Environment

Class Loader
Bytecode Verifier

Java Class Libraries



Java Virtual machine

Operating System

Hardware

Java Features / Buzzwords

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



Simple

- Java was designed to be easy for the professional programmer to learn and use effectively.
- If we already understand the basic concepts of object-oriented programming, learning Java will be even easier.
- If we 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++.
- No Pointers in java.

Secure : Java Programs run inside virtual machine . No pointers in java.

- Java is best know for its security. With java we can develop virus free



Portable

- Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
- Write once, run everywhere

Object Oriented

- Everything in Java is an Object. All program code and data reside within objects and classes.
- Java comes with an extensive set of classes arranged in Packages.



Robust

- Robust means strong. Java is robust because
- It uses strong memory management system. Automatic garbage collection.
- Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Java handles exceptions using Exceptional handling.

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.



Architecture - Neutral

- 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, any time, forever.”

Distributed

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.



Interpreted and High Performance

- 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.
- 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.

Dynamic

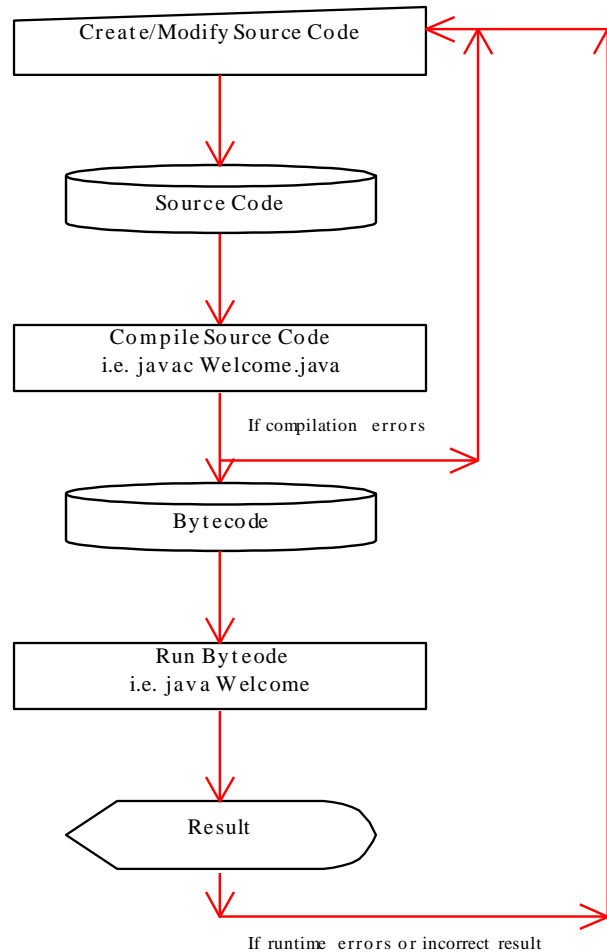
- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.



Creating and Compiling Programs

On command line

`javac file.java`

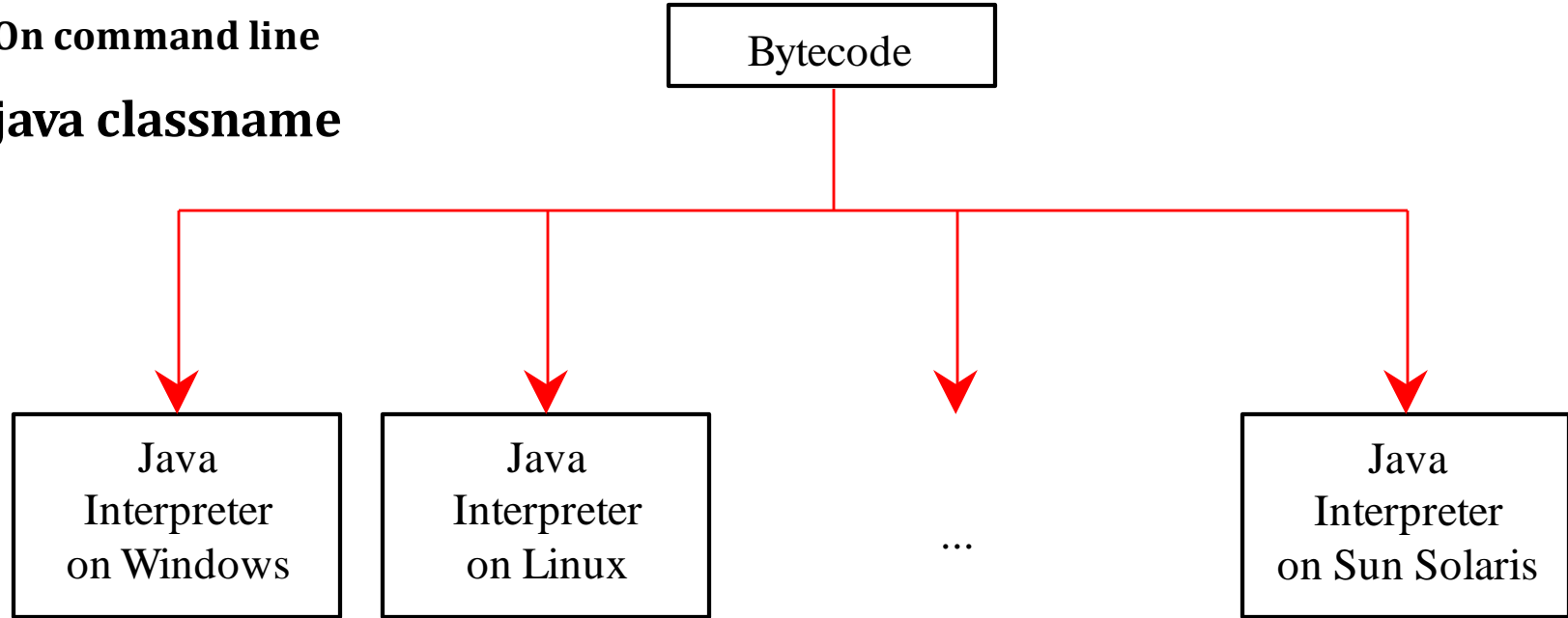




Executing Application

On command line

java classname





Java Virtual Machine

- **Java Virtual Machine (JVM)** is an abstract computing machine.
- **Java Runtime Environment (JRE)** is an implementation of the JVM.
- **Java Development Kit (JDK)** contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.
- **Just In Time compiler (JIT)** is runs after the program has started executing, on the fly. It has access to runtime information and makes optimizations of the code for better performance.

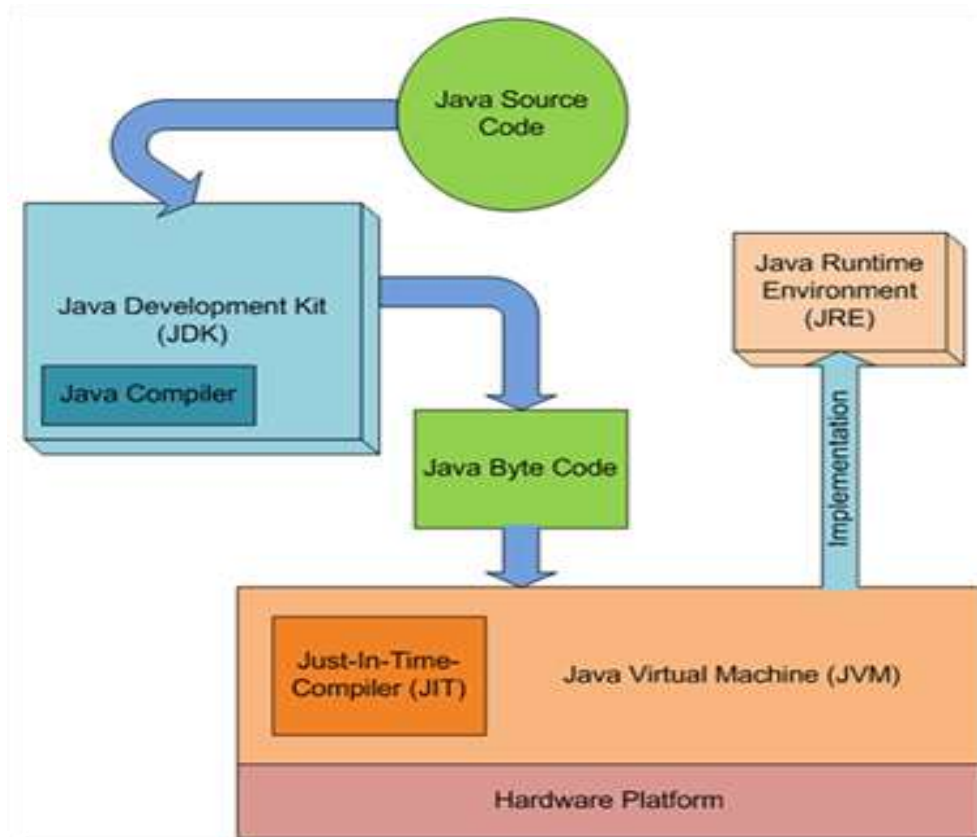


Java Virtual Machine

- JVM becomes an instance of JRE at runtime of a Java program.
- It is widely known as a runtime interpreter.
- The Java virtual machine (JVM) is the cornerstone on top of which the Java technology is built upon.
- It is the component of the Java technology responsible for its hardware and platform independence.
- JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programmes from JDK.



Java Virtual Machine





Difference between C and Java

- Java does not support Struct and Union keywords
- Java does not support user defined data types like typedef and enum.
- Java does not support functions like sizeof() and goto
- In Java we cannot use #include and #define headers
- Java adds a new operators like instanceof and labeled break and continue statements.

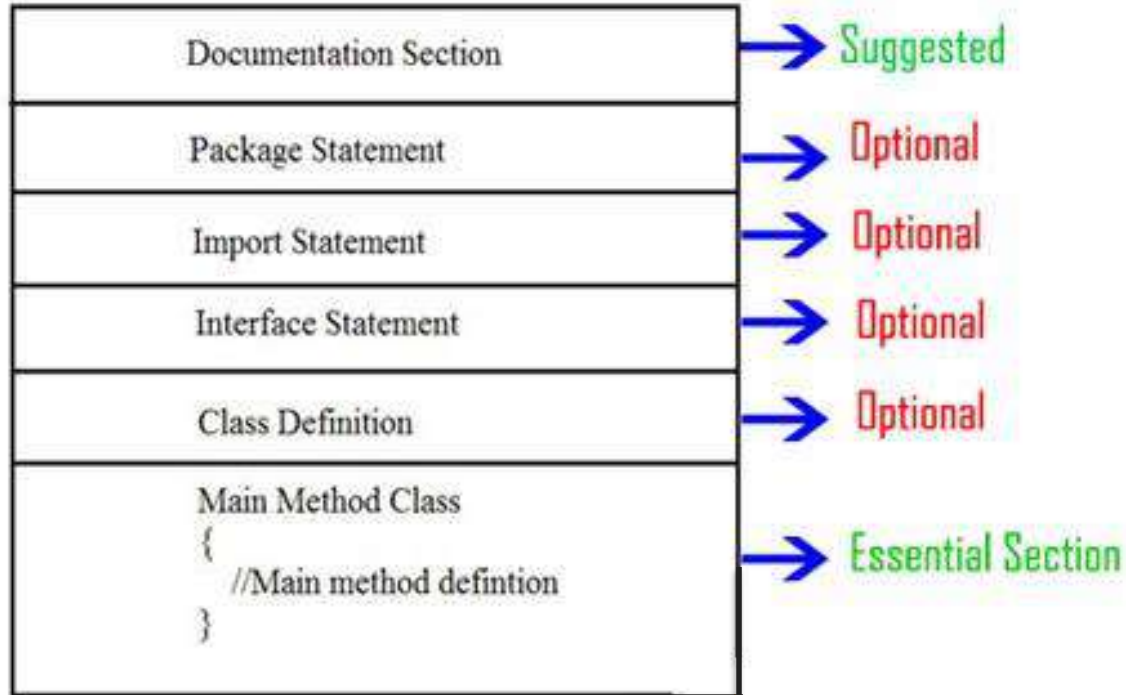


Difference between C++ and Java

- Java is a true object oriented language while C++ is basically C with Object Oriented extension.
- Java does not supports Operator Overloading.
- Java does not support template classes.
- Java does not support pointers
- Java does not support global variables.
- Java does not support multiple Inheritance. This is accomplished using a new feature called “Interfaces”
- Java has replaced the destructor function with a “finalize()” method.



Program Structure





Program Structure

- A Java program consists of different sections.
- Some of them are mandatory but some are optional.
- The optional section can be excluded from the program depending upon the requirements of the programmer.



Documentation Section

- It includes the comments that improve the readability of the program. A comment is a non-executable statement that helps to read and understand a program especially when your programs get more complex.
- A good program should include comments that describe the purpose of the program, author name, date and time of program creation.
- This section is optional and comments may appear anywhere in the program.



Documentation Section

Single line (or end-of line) comment:

It starts with a double slash symbol (//) and terminates at the end of the current line.

Ex: `// Calculate sum of two numbers`

Multiline Comment:

This style of comments can be used on part of a line, a whole line or more commonly to define multi-line comment.

Ex: `/*calculate sum of two numbers and it is a multiline comment*/`

Documentation comments:

Such comments begin with delimiter `/**` and end with `*/`.

Ex: `/** The text enclosed here will be part of program documentation */`



Package Statement

- A package is a collection of classes, interfaces and sub-packages.
A sub package contains collection of classes, interfaces and sub-sub packages etc.
- Ex: package institute;
- This statement declares that all classes and interfaces defined in this source file are part of the institute package.
- Only one package declaration can appear in the source file.



Import statements

- Java contains many predefined classes that are stored into packages.
- An import statement is used for referring classes that are declared in other packages.
- The import statement is written after a package statement but before any class definition.
- You can import a specific class or all the classes of the package.

Ex: `import java.util.Date;`
 `import java.applet.*;`



Interface Section

- An interface is similar to a class but contains only constants and method declarations.
- Interfaces cannot be instantiated.
- They can only be implemented by other interfaces.

```
interface stack
{
    void push(int item); // Insert item into stack
    int pop(); // Delete an item from stack
}
```



Class

Definition

- Java program may contain multiple class definition.
- Classes are primary feature of Java program.
- The classes are used to map real world problems.

```
class Addition
{
    void add(String args[])
    {
        int a=2, b=3, c;
        c=a+b;
        System.out.println(c);
    }
}
```



Main Method Class

Section

- Every program in Java consists of at least one class, the one that contains the main method.
- The main () method which is from where the execution of program actually starts.
- The main method can create objects, evaluate expressions, and invoke other methods and much more.
- On reaching the end of main, the program terminates and control passes back to the operating system.
- The class section is mandatory.



Example

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



- name of class is same as name of file (which has `.java` extension)
 - body of class surrounded by `{ }`
 - this class has one method called `main`
1. all Java applications must have a main method in one of the classes
 2. execution starts here
 3. body of method within `{ }`
- all other statements end with semicolon `;`



Java keywords

- **public**
 - visibility could be **private**
- **static**
 - the **main** method belongs to the **Hello** class, and not an instance (object) of the class
- **void**
 - method does not return a value



Compile Welcome.java

javac Welcome.java

Output: Welcome.class

Run

java Welcome

Output: Welcome to Java



Installation of JDK

Step 0: Un-Install Older Version(s) of JDK/JRE

Step 1: Download JDK

Goto Java SE download site @

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Step 2: Install JDK

Run the downloaded installer

Step 3: Include JDK's "bin" Directory in the PATH

Step 4: Verify the JDK Installation

Step 5: Write a Hello-World Java Program

Step 6: Compile and Run the Hello-World Java Program



Java Programming

Unit – 1 Programming Concepts

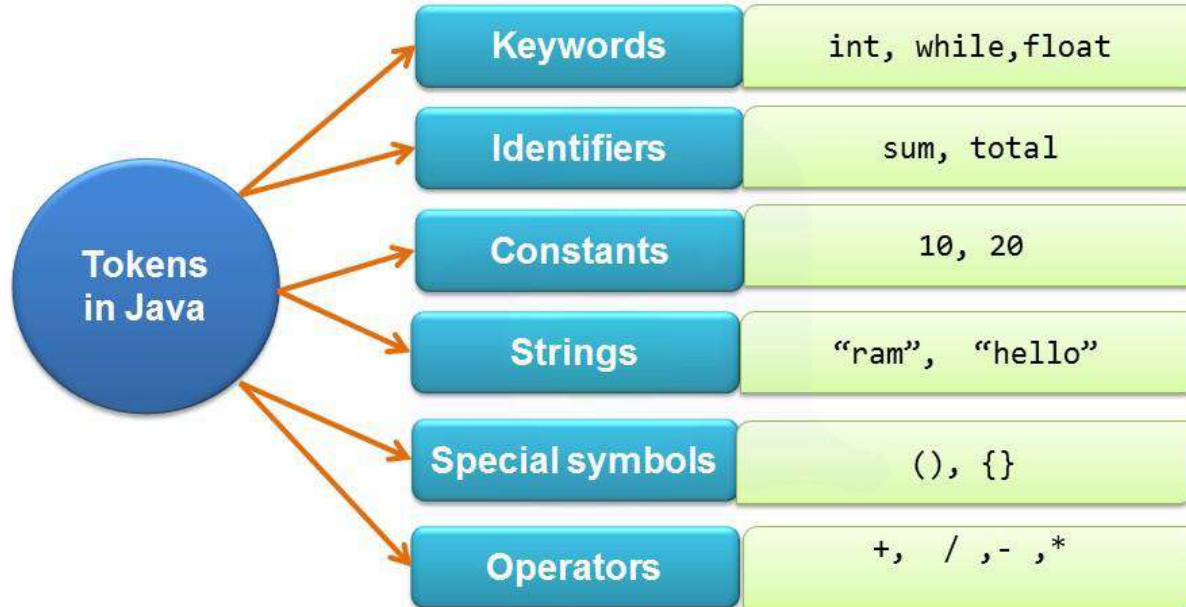




TABLE OF CONTENTS

1. Java Tokens
2. Control Structures
3. Classes and Objects
4. Garbage Collection

Java Tokens





Variables

- A *variable* is a location in memory that can hold values of a certain *data type*
- Each variable must be *declared* before it is used
- The declaration allocates a location in memory to hold values of this type
- Variable types can be
 - primitive
 - reference to an object



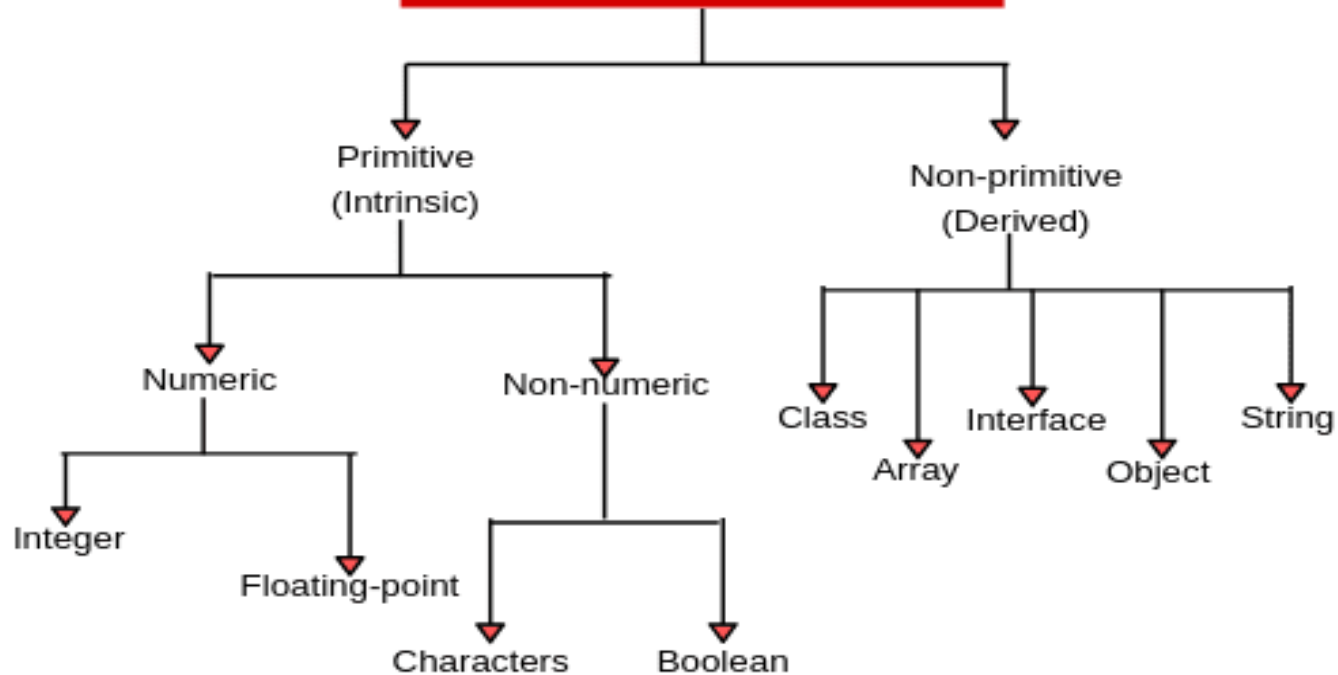
Variables

- we have to declare variables before we use them
- unlike C, variables can be declared anywhere within block
- use meaningful names
- start with lower case, capitalise first letter of subsequent words

```
public class VariableExample {  
    public static void main(String[] args){ int x;  
        x = 3;  
        System.out.println(x); x = 4;  
        System.out.println(x);  
    }  
}
```




Data Types in Java





Data types

int

4 byte integer (whole number)

range -2147483648 to +2147483648

float

4 byte floating point number

decimal points, numbers outside range of **int**

double

8 byte floating point number

15 decimal digits (float has 7) so bigger precision and range

char

2 byte letter

String

string of letters

boolean

true or **false** (not 1 or 0)



Primitive Data Types

- A *data type* is defined by a set of values and the operators you can perform on them
- The Java language has several predefined types, called *primitive data types*
- The following reserved words represent the eight different primitive data types:
 - **byte, short, int, long, float, double, boolean, char**



Integers

- There are four integer data types. They differ by the amount of memory used to store them

Type	Bits	Value Range
byte	8	-127 ... 128
short	16	-32768 ... 32767
int	32	about 9 decimal digits
long	64	about 18 decimal digits



Floating Point

There are two floating point types

Type	Bits	Range (decimal digits)	Precision (decimal digits)
float	32	38	7
double	64	308	15



Characters

- A **char** value stores a single character from the *Unicode character set*
- A *character set* is an ordered list of characters

'A', 'B', 'C', ... , 'a', 'b', ... , '0', '1', ... , '\$', ...

Boolean

- A **boolean** value represents a true/false condition.
- It can also be used to represent any two states, such as a light bulb being on or off
- The reserved words **true** and **false** are the only valid values for a boolean type



Datatypes

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0



Identifiers: Syntax

- *Identifiers* are the words a programmer uses in a program
- Identifier syntactic rules:
 - Can be made up of any length of
 - letters
 - digits
 - underscore character (`_`)
 - dollar sign (`$`)
 - Cannot begin with a digit
- Java is *case sensitive*
 - User and user are completely different identifiers



Naming Conventions for Identifiers

- **Classes**
 - Names given in lowercase except for first letter of each word in the name
- **Variables**
 - Same as classes, except first letter is lowercase
- **Constants**
 - All caps with _ between words
- **Methods**
 - like variable names but followed by parentheses



Keywords

- Keywords are predefined, reserved words used in Java programming that have special meanings to the compiler.
 - Ex: `int score;`
- Here `int` is a keyword. It indicates that the variable `score` is of `Integer` type.
- These are predefined words by Java so it cannot be used as a variable or object name.



Keywords

Complete List of Java Keywords

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



Literals – Examples

Integers	-	4, 19, -5, 0, 1000
Doubles	-	3.14, 0.0, -16.123
Strings	-	"Hi Mom" "Enter the number : "
Character	-	'A' 'X' '9' '\$' '\n'
Boolean	-	true, false



Constants

- We may declare that a variable is a *constant* and its value may never change.

```
final double PI = 3.14159;  
final int CHINA_OLYMPICS_YEAR = 2008;
```

- Advantages:
 - readability
 - efficiency
 - error detection



Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If either or both operands used by an arithmetic operator are floating point, then the result is a floating point



Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

14 / 3 equals 4

8 / 12 equals 0

- The remainder operator (%) returns the remainder after dividing the second operand into the first

14 % 3 equals 2

8 % 12 equals 8



Operator Precedence

- Operators can be combined into complex expressions

`result = total + count / max - offset;`

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order



Operator Precedence Examples

Expression	Result
$10 - 7 - 1$	2
$10 - (7 - 1)$	4
$1 + 2 * 3$	7
$(1 + 2) * 3$	9
$1 - 2 * 3 + 4 * 5$	15



Arithmetic Operators

- An *operator* is a mapping that maps one or more values to a single value:
- Binary Operators:
 - $a + b$ adds a and b
 - $a - b$ subtracts b from a
 - $a * b$ multiplies a and b
 - a / b divides a by b
 - $a \% b$ the remainder of dividing a by b
- Unary Operator:
 - $-a$ The negation of a



Simple Arithmetic

```
public class Example {  
    public static void main(String[] args) {  
        int j, k, p, q, r, s, t;  
        j = 5;  
        k = 2;  
        p = j + k;  
        q = j - k;  
        r = j * k;  
        s = j / k;  
        t = j % k;  
        System.out.println("p = " + p);  
        System.out.println("q = " + q);  
        System.out.println("r = " + r);  
        System.out.println("s = " + s);  
        System.out.println("t = " + t);  
    }  
}
```

```
> java Example  
p = 7  
q = 3  
r = 10  
s = 2  
t = 1  
>
```



Shorthand Operators $+=$, $-=$, $*=$, $/=$, $\%=$

Common

$a = a + b;$

$a = a - b;$

$a = a * b;$

$a = a / b;$

$a = a \% b;$

Shorthand

$a += b;$

$a -= b;$

$a *= b;$

$a /= b;$

$a \% = b;$



Shorthand Operators

```
public class Example {  
    public static void main(String[] args) {  
        int j, p, q, r, s, t;  
        j = 5;  
        p = 1; q = 2; r = 3; s = 4; t = 5;  
        p += j;  
        q -= j;  
        r *= j;  
        s /= j;  
        t %= j;  
        System.out.println("p = " + p);  
        System.out.println("q = " + q);  
        System.out.println("r = " + r);  
        System.out.println("s = " + s);  
        System.out.println("t = " + t);  
    }  
}
```

```
> java Example  
p = 6  
q = -3  
r = 15  
s = 0  
t = 0  
>
```



Shorthand Increment and Decrement ++ and --

Common

`a = a + 1;`

`a = a - 1;`

Shorthand

`a++; or ++a;`

`a--; or --a;`



Increment and Decrement

```
public class Example {  
    public static void main(String[] args) {  
        int j, p, q, r, s;  
        j = 5;  
        p = ++j; // j = j + 1; p = j;  
        System.out.println("p = " + p);  
        q = j++; // q = j;    j = j + 1;  
        System.out.println("q = " + q);  
        System.out.println("j = " + j); r  
        = --j; // j = j - 1;    r = j;  
        System.out.println("r = " + r); s  
        = j--; // s = j;    j = j - 1;  
        System.out.println("s = " + s);  
    }  
}
```

```
> java example  
p = 6  
q = 6  
j = 7  
r = 6  
s = 6  
>
```



Relational Operators

> < >= <= == !=

Primitives

- Greater Than >
- Less Than <
- Greater Than or Equal >=
- Less Than or Equal <=

Primitives or Object References

- Equal (Equivalent) ==
- Not Equal !=

The Result is Always true or false



Relational Operator Examples

```
public class Example {  
    public static void main(String[] args) {  
        int p = 2; int q = 2; int r = 3;  
        Integer i = new Integer(10);  
        Integer j = new Integer(10);  
  
        System.out.println("p < r " + (p < r));  
        System.out.println("p > r " + (p > r));  
        System.out.println("p == q " + (p == q));  
        System.out.println("p != q " + (p != q));  
  
        System.out.println("i == j " + (i == j));  
        System.out.println("i != j " + (i != j));  
    }  
}
```

```
> java Example  
p < r true  
p > r false  
p == q true  
p != q false  
i == j false  
i != j true  
>
```



Logical Operators (boolean)

&& || !

- Logical AND &&
- Logical OR |
- Logical NOT !



Assignment Statements

- An assignment statement takes the following form

variable-name = expression;

- The expression is first *evaluated*
- Then, the result is stored in the variable, overwriting the value currently stored in the variable



Logical Operators (Bit Level) &

| \wedge \sim

- AND &
- OR |
- XOR \wedge
- NOT \sim



Logical (bit) Operator Examples

```
public class Example {  
    public static void main(String[] args) {  
        int a = 10;    // 00001010 = 10  
        int b = 12;    // 00001100 = 12  
        int and, or, xor, na;  
        and = a & b;    // 00001000 = 8  
        or = a | b;    // 00001110 = 14  
        xor = a ^ b;    // 00000110 = 6  
        na = ~a;        // 11110101 = -11  
        System.out.println("and " + and);  
        System.out.println("or " + or);  
        System.out.println("xor " + xor);  
        System.out.println("na " + na);  
    }  
}
```

```
> java Example  
and 8  
or 14  
xor 6  
na -11  
>
```



Shift Operators (Bit Level)

<< >> >>>

- Shift Left << Fill with Zeros
- Shift Right >> Based on Sign
- Shift Right >>> Fill with Zeros



<<

[illegible]

>>

```
a          00000000000000000000000000000000000011      3
a >> 2    00000000000000000000000000000000000000      0

b          11111111111111111111111111111111111100     -4
b >> 2    11111111111111111111111111111111111111     -1
```



>>>

```
int a = 3; // ...00000011 = 3
int b = -4; // ...11111100 = -4
```

Right 0

```

a          0000000000000000000000000000000000011      3
a >>> 2  00000000000000000000000000000000000000      0

b          11111111111111111111111111111111111100     -4
b >>> 2  00111111111111111111111111111111111111      +big

```




Shift Operator Examples

```
public class Example {  
    public static void main(String[] args) {  
        int a = 3;    // ...00000011 = 3  
        int b = -4;   // ...11111100 = -4  
        System.out.println("a<<2 = " + (a<<2));  
        System.out.println("b<<2 = " + (b<<2));  
        System.out.println("a>>2 = " + (a>>2));  
        System.out.println("b>>2 = " + (b>>2));  
        System.out.println("a>>>2 = " + (a>>>2));  
        System.out.println("b>>>2 = " + (b>>>2));  
    }  
}
```

```
> java Example  
a<<2 = 12  
b<<2 = -16  
a>>2 = 0  
b>>2 = -1  
a>>>2 = 0  
b>>>2 = 1073741823  
>
```



Ternary Operator

? :

Any expression that evaluates to a boolean value.

boolean_expression ? expression_1 : expression_2

If **true** this expression is evaluated and becomes the value entire expression.

If **false** this expression is evaluated and becomes the value entire expression.



Ternary (? :) Operator Examples

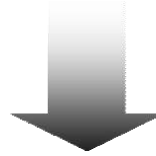
```
public class Example {  
    public static void main(String[] args) {  
        boolean t = true;  
        boolean f = false;  
  
        System.out.println("t?true:false "+(t ? true : false ));  
        System.out.println("t?1:2 "+(t ? 1 : 2 ));  
        System.out.println("f?true:false "+(f ? true : false ));  
        System.out.println("f?1:2 "+(f ? 1 : 2 ));  
    }  
}
```

```
> java Example  
t?true:false true  
t?1:2 1  
f?true:false false  
f?1:2 2  
>
```



String (+) Operator (Automatic Conversion with Primitives)

"The number is " + 4



"The number is " + "4"



"The number is 4"



Type Conversion

- Widening numeric conversions
 - int to long, float, or double
 - long to float or double
 - float to double
- Narrowing numeric conversions
 - int to byte, short, or char
 - long to int
 - float to int or long
 - double to int, long, or float



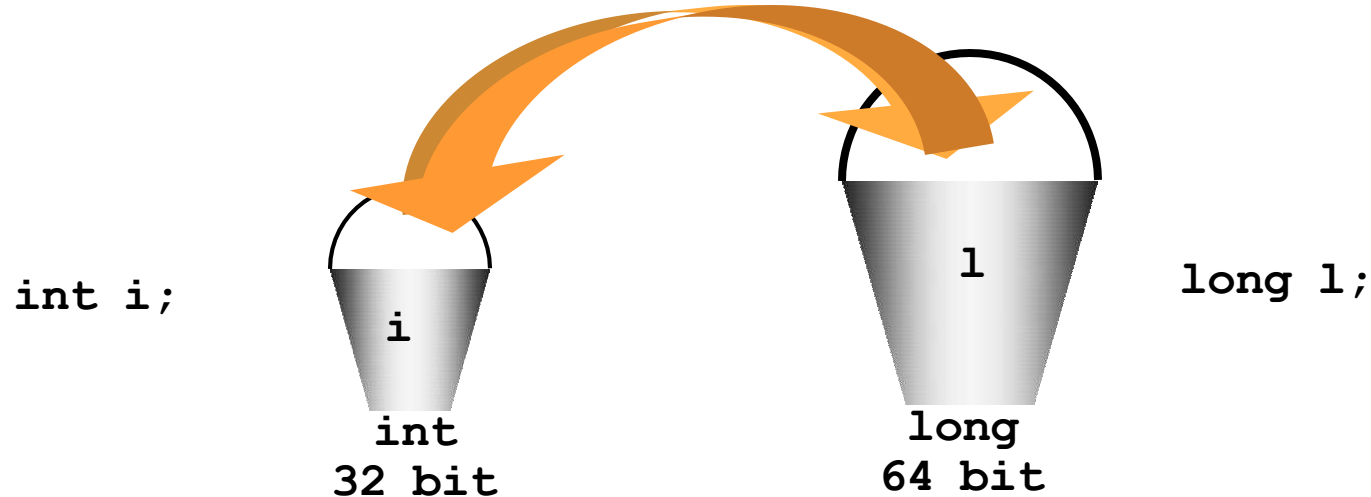
Moving Between Bucket "Casting"

```
i = (int)l;
```

Narrowing

```
l = i;
```

Widening





Type Casting

Implicit casting

`double d = 3; (type widening)`

Explicit casting

`int i = (int)3.0; (type narrowing)`



Flow of Control

- Java executes one statement after the other in the order they are written
- Many Java statements are flow control statements:

Alternation : if, if else, Nested If, Else If, switch

Looping : while, do while, for

Escapes : break, continue, return

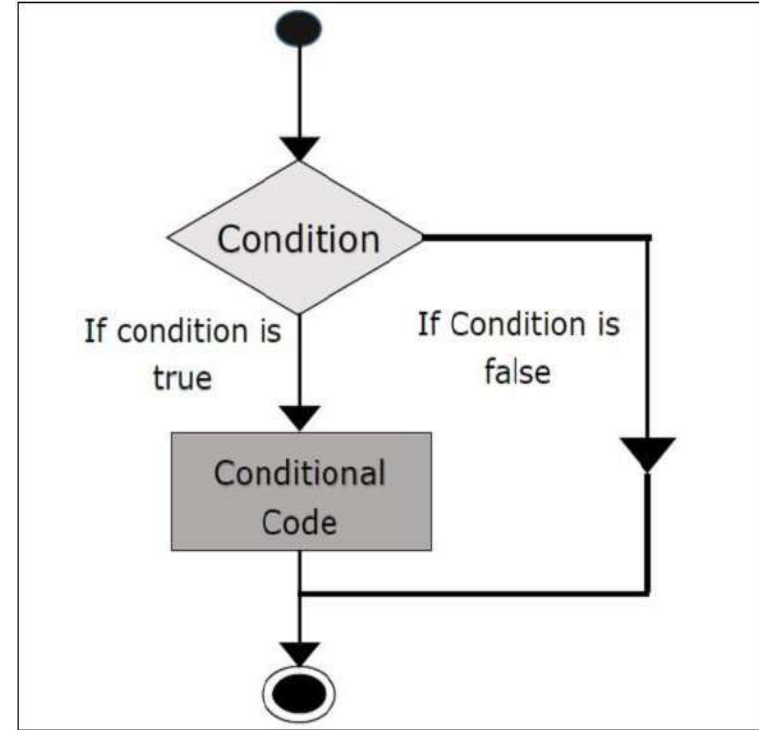


Simple If

The if statement evaluates an expression and if that evaluation is true then the specified action is taken

if (x < 10)

x = 10;

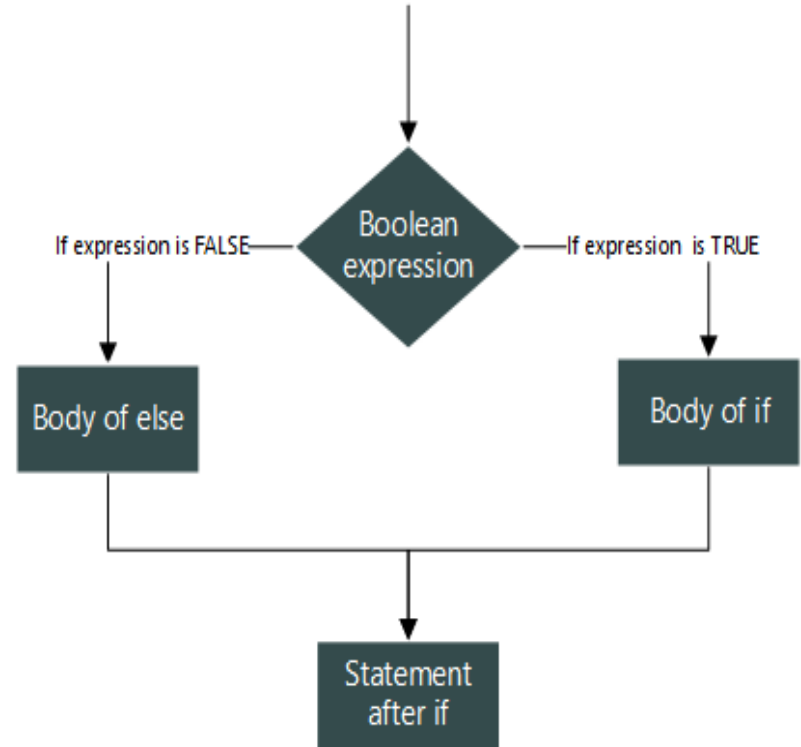




If... else

The if ... else statement evaluates an expression and performs one action if that evaluation is true or a different action if it is false.

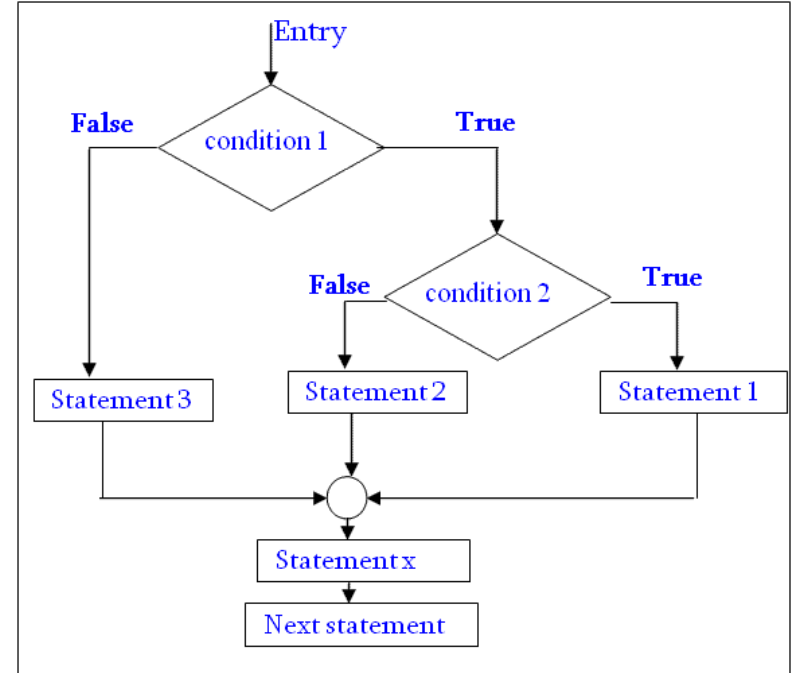
```
if (x != oldx) {  
    System.out.print("x was changed");  
}  
else {  
    System.out.print("x is unchanged");  
}
```





Nested if ... else

```
if ( myVal > 100 )  
{  
    if ( remainderOn == true )  
    {  
        myVal = mVal % 100;  
    }  
    else {  
        myVal = myVal / 100.0;  
    }  
}  
else  
{  
    System.out.print("myVal is in range");  
}
```

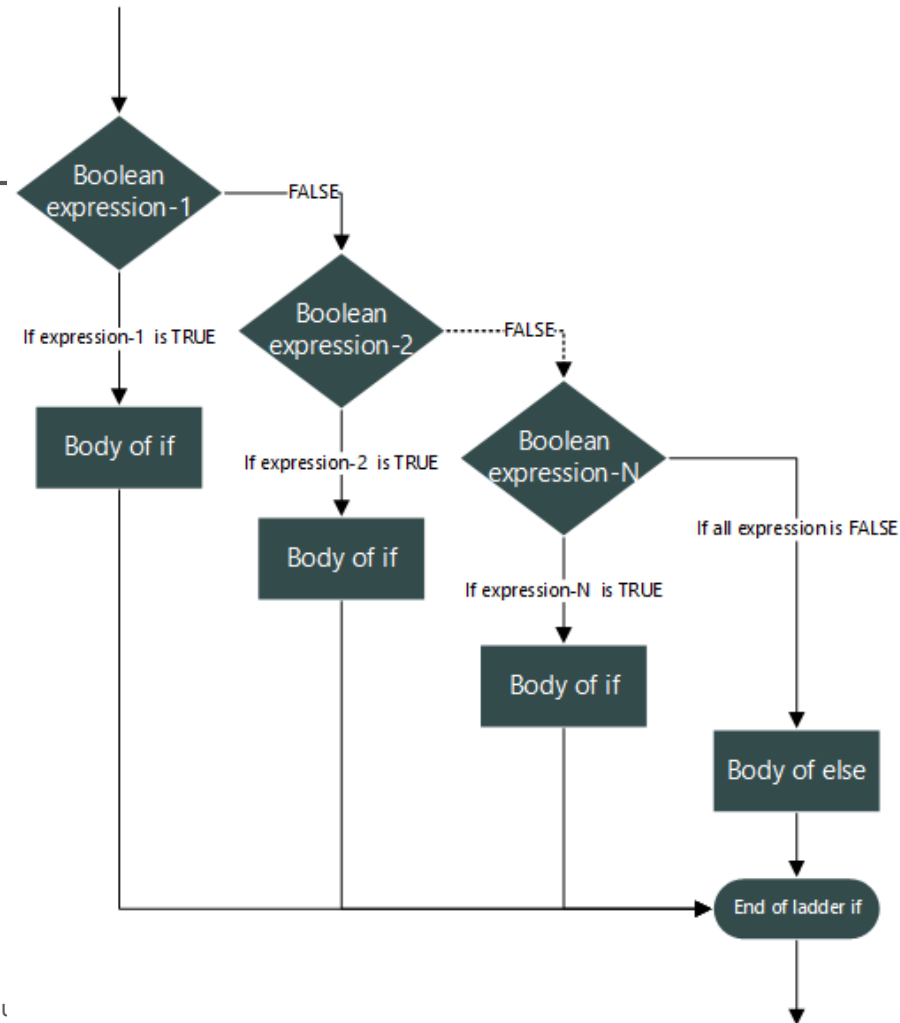




Else If Ladder

Useful for choosing between alternatives:

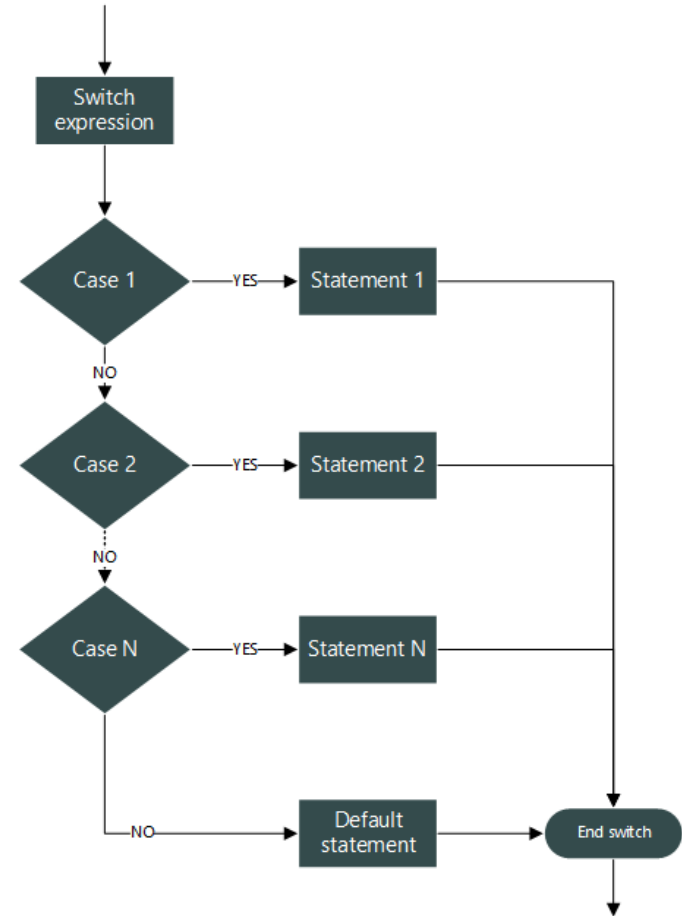
```
if ( n == 1 )  
{  
    // execute code block #1  
}  
else if ( j == 2 ) {  
    // execute code block #2  
}  
else {  
    // if all previous tests have failed,  
    execute code block #3  
}
```





The switch Statement

```
switch ( n ) {  
    case 1:  
        // execute code block #1  
        break;  
  
    case 2:  
        // execute code block #2  
        break;  
  
    default:  
        // if all previous tests fail then  
        // execute code block  
        break;  
}
```





while loop

```
int i=0;
```

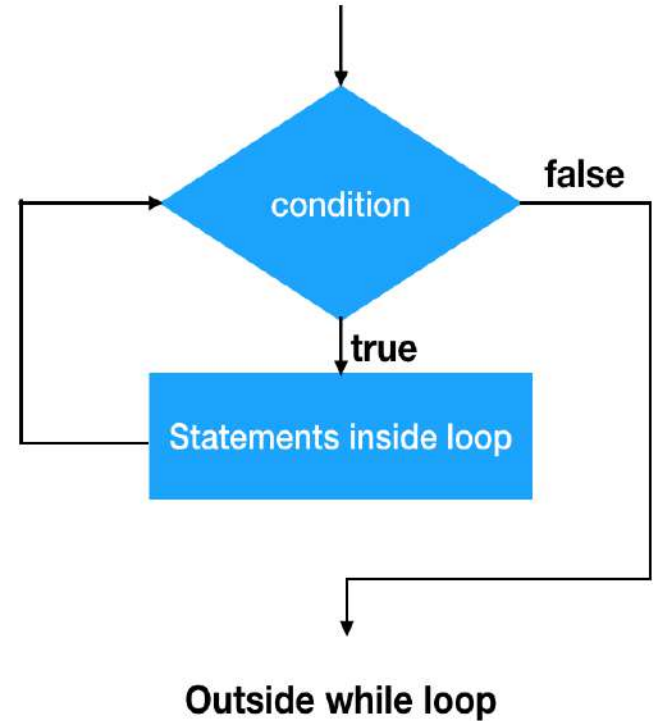
```
while(i<10)
```

```
{
```

```
    System.out.println(i);
```

```
    i++;
```

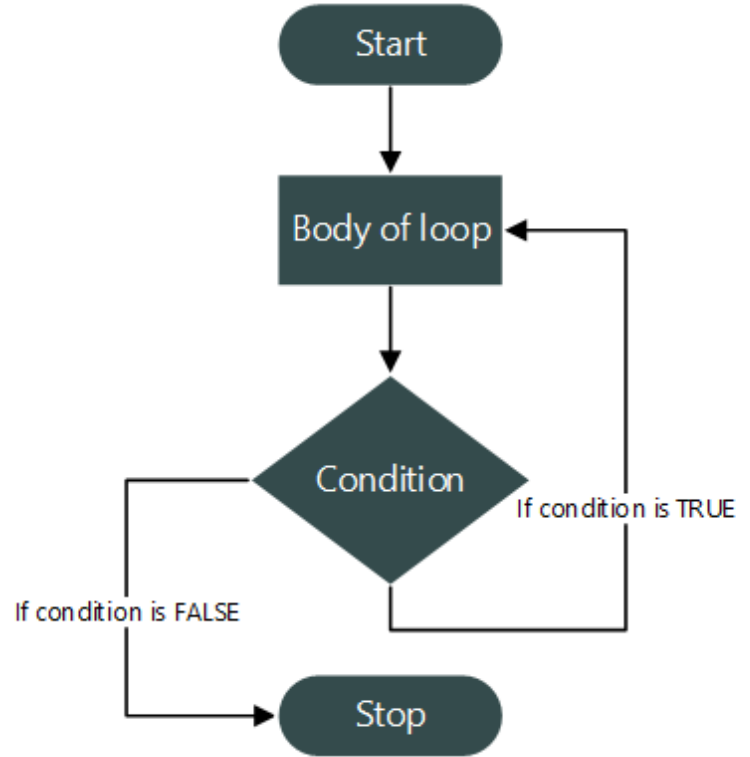
```
}
```





do {... } while loops

```
int i=0;  
do  
{  
    System.out.println(i);  
    i++;  
} while(i<10);
```





The for loop

Loop n times

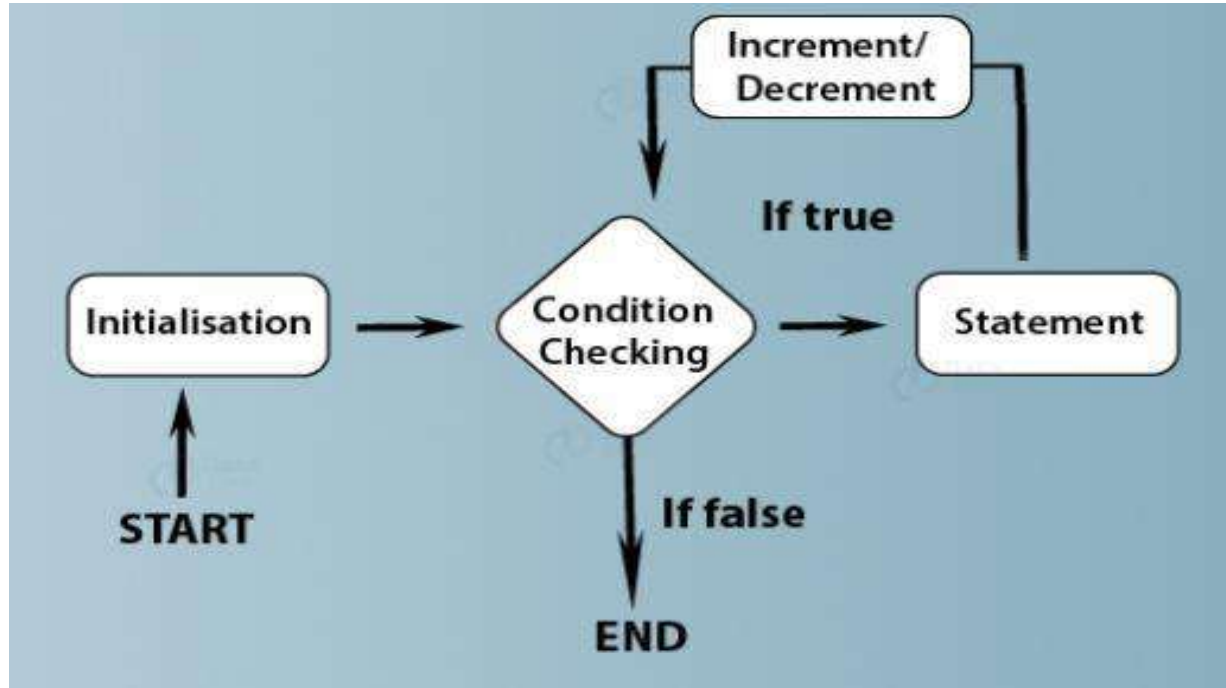
```
for ( i = 0; i < n; n++ ) {  
    // this code body will execute n times  
    // i from 0 to n-1  
}
```

Nested for:

```
for ( j = 0; j < 10; j++ ) {  
    for ( i = 0; i < 20; i++ ){  
        // this code body will execute 200 times  
    }  
}
```




For loops





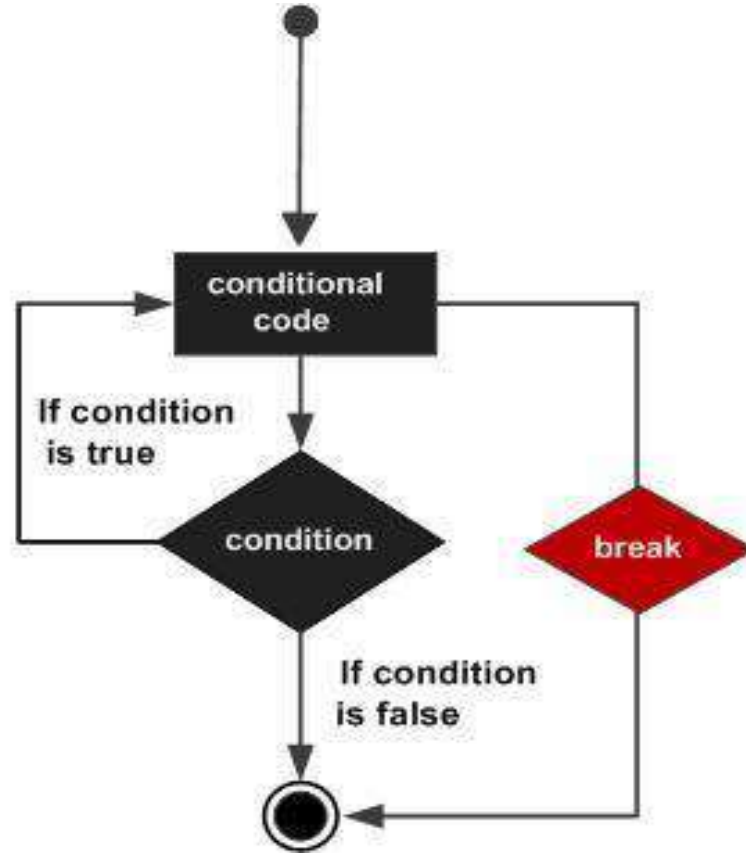
Break

- A break statement causes an exit from the **innermost** containing **while**, **do**, **for** or **switch** statement.

```
for ( int i = 0; i < 10, i++ ){  
    if ( i == 3 )  
    {  
        break;  
    }  
    System.out.println(i);  
}
```



Break





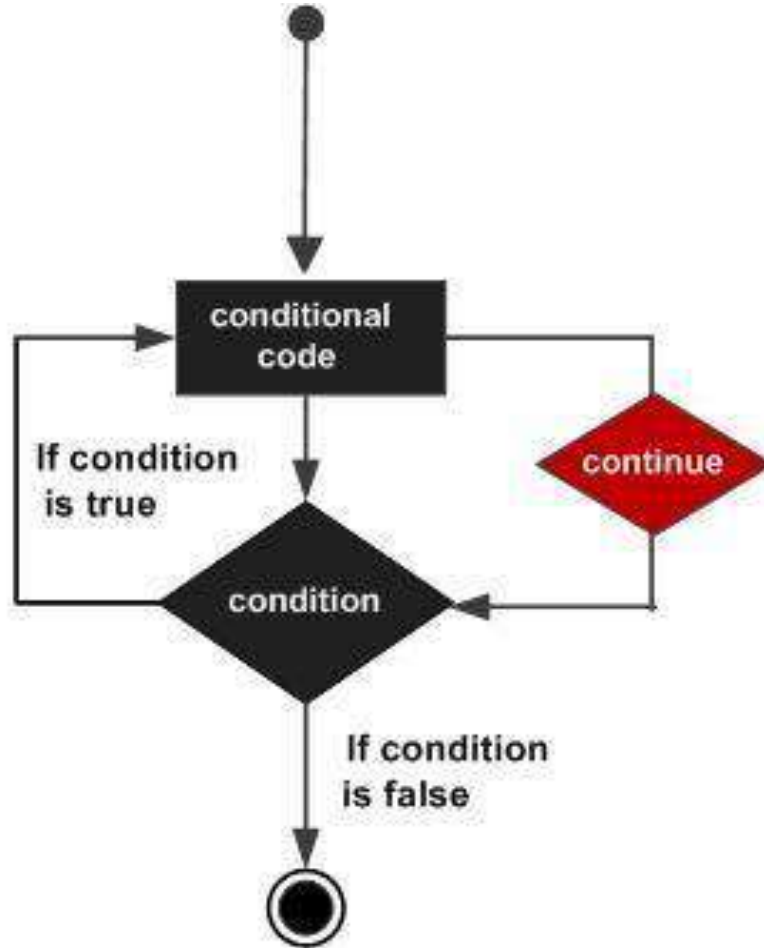
Continue

- Can only be used with while, do or for.
- The continue statement causes the innermost loop to start the next iteration immediately

```
for ( int i = 0; i < 10; i++ ) {  
    if ( i==3 ) continue;  
  
    System.out.println(i);  
}
```



Continue





Labeled break and continue

- Labeled block
 - Set of statements enclosed by {}
 - Preceded by a label
- Labeled break statement
 - Exit from nested control structures
 - Proceeds to end of specified labeled block
- Labeled continue statement
 - Skips remaining statements in nested-loop body
 - Proceeds to beginning of specified labeled block



Labeled Continue example

```
public class LabelledContinue
{
    public static void main( String args[] )
    {
        stop:
            for ( int r = 1; r <= 10; r++ )
            {
                for ( int c = 1; c <= 5 ; c++ )
                {
```

```
                    if ( r == 5 )
                        continue stop;
                    System.out.print(c);
                }
                System.out.println();
            }
        }
    }
```



Labelled break example

```
public class LabelledContinue
{
    public static void main( String args[] )
    {
        stop:
            for ( int r = 1; r <= 10; r++ )
            {
                for ( int c = 1; c <= 5 ; c++ )
                {
```

```
                    if ( r == 5 )
                        break stop;
                    System.out.print(c);
                }
                System.out.println();
            }
        }
    }
```




Streams

- A **Stream** is a continuous, seemingly infinite supply of or sink for data
- An **ordered** sequence of bytes flows in a specified direction
 - in from some source
 - can be sent out to some destination
- Acts as a pipe connected to your program
- A channel providing communication to and from the outside world
 - Standard Input: `System.in`
 - Standard Output: `System.out`



System output

- Java provides print methods in the class `System.out`

`println(name);`

- prints out what is stored in *name*, then goes to a new line

`print(name);`

- prints out what is stored in *name*, but does not start a new line

`print("My name is " + name);`

- put text in quotes
- use + to print more than one item



Program Input

- For now, we'll get input from two sources
- Standard Input Stream
 - Scanner class
- Command Line Arguments



Scanner

- Scanner is a class that reads data from the standard input stream and parses it into tokens based on a **delimiter**
- A delimiter is a character or set of characters that distinguish one token from another
- By default the Scanner class uses white space as the delimiter
- The tokens can be read in either as Strings

`next()`

`nextLine()`

- Or they can be read as primitive types

Ex: `nextInt()`, `nextFloat()`, `nextDouble()`

Scanner Class

Example

```
import java.util.Scanner;
class scan
{
    public static void main(String ar[])
    {
        Scanner s=new Scanner(System.in);
        System.out.print("Enter A :");
        int a=s.nextInt();
        System.out.print("Enter B :");
        int b=s.nextInt();
        System.out.println("\n After Adding A & B :"+(a+b));
    }
}
```

The image features a white background with abstract geometric patterns in the corners. These patterns are composed of various colored triangles (red, orange, teal, and dark blue) arranged in a way that creates a 3D effect, resembling stacked cubes or a complex crystalline structure. The triangles are of different sizes and are scattered around the central text, with some appearing to float or be part of a larger assembly.

End of Unit I