What Is Java?

New object-oriented programming (OOP) language developed by SUN Microsystems

Similar to C and C++, except without some of the confusing, poorly understood features of C++

Extensive networking facilities

Extensive set of APIs for GUIs, distributed computing, 2D/3D graphics, mail, and others

Portable: Write Once, Run Anywhere

Multithreading support built into the language

Features Removed From C/C++

No typedefs, defines or preprocessor

No header files

No structures or unions

No enums

No functions - only methods in classes

No multiple inheritance

No goto

No operator overloading (except "+" for string concatenation)

No automatic type conversions (except for primitive types)

No pointers

Java Virtual Machine

Java is compiled into bytecodes

Bytecodes are high-level, machine-independent instructions for a hypothetical machine, the Java Virtual Machine (JVM)

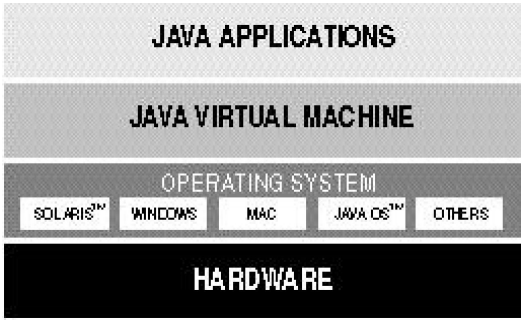
The Java run-time system provides the JVM

The JVM interprets the bytecodes during program execution

Since the bytecodes are interpreted, the performance of Java programs slower than comparable C/C++ programs

But the JVM is continually being improved and new techniques are achieving speeds comparable to native C++ code

Java Virtual Machine



All Java programs run on top of the JVM

The JVM was first implemented inside Web browsers, but is now available on a wide variety of platforms

The JVM interprets the bytecodes defined in a machine-independent binary file format called a *class* file.

Types Of Java Programs

Application

Standalone Java program that can run independent of any Web browser

Applet

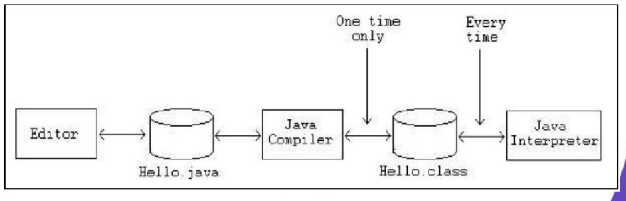
Java program that runs within a Java-enabled Web browser

Servlet

Java software that is loaded into a Web server to provide additional server functionality CGI programs

Phases of a Java Program

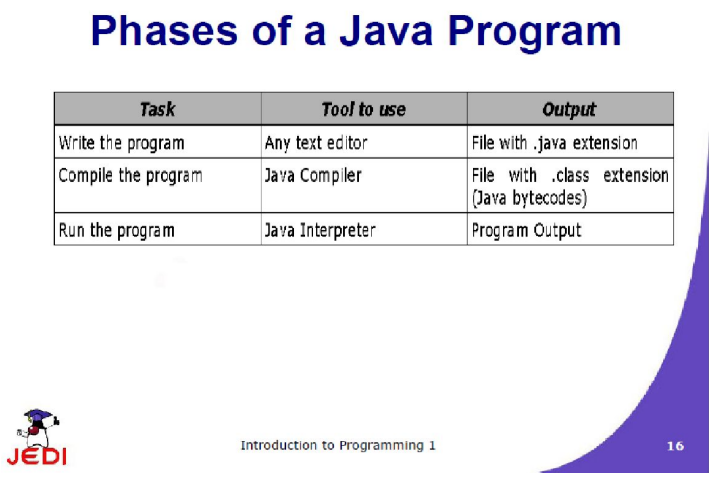
- The following figure describes the process of compiling and executing a Java program



JEDI

Introduction to Programming 1

15



Java Keywords:-

Keyword	Meaning
abstract an	abstract class or method
assert	used to locate internal program errors
Boolean	the Boolean type
break	breaks out of a switch or loop
byte	the 8-bit integer type
case	a case of a switch
catch	the clause of a try block catching an exception
char	the Unicode character type
class	defines a class type
const	not used
continue	continues at the end of a loop
default	the default clause of a switch
do	the top of a do/while loop

double	the double-precision floating-number type
else	the else clause of an if statement
extends	defines the parent class of a class
final	a constant, or a class or method that cannot be overridden
finally	the part of a try block that is always executed
float	the single-precision floating-point type
for	a loop type
goto	not used
if	a conditional statement
implements	defines the interface(s) that a class implements
import	imports a package
instanceof	tests if an object is an instance of a class
int	the 32-bit integer type
interface	an abstract type with methods that a class can implement
long	the 64-bit long integer type
native	a method implemented by the host system
new	allocates a new object or array
null	a null reference
package	a package of classes
private	a feature that is accessible only by methods of this class
protected	a feature that is accessible only by methods of this class, its children, and other classes in the same package
public	a feature that is accessible by methods of all classes
return	returns from a method
short	the 16-bit integer type

static	a feature that is unique to its class, not to objects of its class
strictfp	Use strict rules for floating-point computations
super	the superclass object or constructor
switch	a selection statement
synchronized	a method or code block that is atomic to a thread
this	the implicit argument of a method, or a constructor of this class
throw	throws an exception
throws	the exceptions that a method can throw
transient	marks data that should not be persistent
try	a block of code that traps exceptions
void	denotes a method that returns no value
volatile	ensures that a field is coherently accessed by multiple threads
while	a loop

History of Java

Around 1990 James Gosling , Bill Joy and others at Sun Microsystems began developing a language called Oak. The wanted it primarily to control microprocessors embedded in consumer items such as cable set-top boxes,VCR's, toasters, and also for personal data assistants (PDA).

To serve these goals, Oak needed to be:

Platform independent (since multiple manufacturers involved)

Extremely reliable

Compact.

However, as of 1993, interactive TV and PDA markets had failed to take off. Then the Internet and Web explosion began, so Sun shifted the target market to Internet applications and changed the name of the project to Java.

By 1994 Sun's HotJava browser appeared. Written in Java in only a few months, it illustrated the power of applets, programs that run within a browser, and also the capabilities of Java for speeding program development.

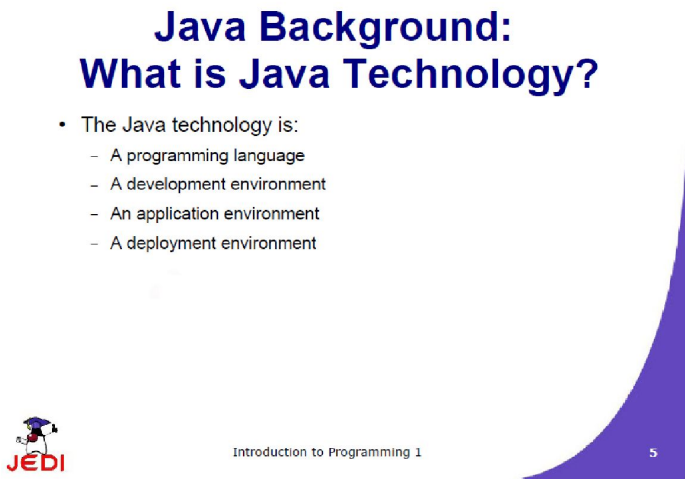
Riding along with the explosion of interest and publicity in the Internet, Java quickly received widespread recognition and expectations grew for it to become the dominant software for browser and consumer applications.

However, the early versions of Java did not possess the breadth and depth of capabilities needed for client (i.e. consumer) applications. For example, the graphics in Java 1.0 seemed crude and clumsy compared to mature software developed with C and other languages.

Applets became popular and remain common but don't dominate interactive or multimedia displays on web pages. Many other "plug-in" types of programs also run within the browser environment.

So Java has not succeeded at development of consumer applications. However, Java's capabilities grew with the release of new and expanded versions (see below) and it became a very popular language for development of enterprise, or middleware, applications such as on line web stores, transactions processing, database interfaces, and so forth.

Java has also become quite common on small platforms such as cell phones and PDAs. Java is now used in several hundred cell phone models. Over 600 million JavaCards, smart cards with additional features provided by Java, have been sold as of the summer of 2004.



What is java

This section introduces you the Java programming language. These days Java programming language is being used for programming web applications. It is also widely used for mobile and electronic items.

1. **What is Java?**

Java is a high-level object-oriented programming language developed by the Sun

Microsystems. Though it is associated with the World Wide Web but it is older than the origin of Web.

2. **Java as an Internet Language</B< a>**

Java is an object oriented language and a very simple language. Because it has no space for complexities. At the initial stages of its development it was called as OAK. OAK was designed for handling set up boxes and devices.

3. **Java as general purpose language**

Java is an Object oriented application programming language developed by Sun Microsystems.

Java Features

4. **Case sensitivity**

What is case sensitivity: Case sensitivity is the mechanism in which words can be differ in meaning based on different use of uppercase and lowercase letters.

5. **Java is Simple and platform Independent**

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language.

6. **Java Enabled browsers**

Java language is the most powerful language and is widely used in the web application. Today most of the web browser are java compatible.

Java Tools

7. **Java Compiler**

To commence with Java programming, we must know the significance of Java Compiler.

8. **Java Interpreter**

We can run Java on most platforms provided a platform must has a Java interpreter.

9. **Java Debugger**

Java debugger helps in finding and the fixing of bugs in Java language programs.

10. **Java Header File Generator**

In Java programming we need to implement some native methods. To implement these methods **Javah** generates C header and source files that are used by C programs to reference an Object's instance variables from native source code.

11. **JavaDoc**

This tool is used to generate API documentation into HTML format from Java source

code.

12. **Applet Viewer**

Applet viewer is a command line program to run Java applets.

Features of JAVA

Platform Independent

The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is more closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM.

Simple

There are various features that makes the java as a simple language. Programs are easy to write and debug because java does not use the pointers explicitly. It is much harder to write the java programs that can crash the system but we can not say about the other programming languages. Java provides the bug free system due to the strong memory management. It also has the automatic memory allocation and deallocation system.

Object Oriented

To be an Object Oriented language, any language must follow at least the four characteristics.

- Inheritance : It is the process of creating the new classes and using the behavior of the existing classes by extending them just to reuse the existing code and adding the.
- additional features as needed
- Encapsulation: : It is the mechanism of combining the information and providing the abstraction.
- Polymorphism: : As the name suggest one name multiple form, Polymorphism is the way of providing the different functionality by the functions having the same name based on the signatures of the methods.
- Dynamic binding : Sometimes we don't have the knowledge of objects about their specific types while writing our code. It is the way of providing the maximum functionality to a program about the specific type at runtime.

As the languages like Objective C, C++ fulfills the above four characteristics yet they are not fully object oriented languages because they are structured as well as object oriented languages. But in case of java, it is a fully Object Oriented language because object is at the outer most level of data structure in java. No stand alone methods, constants, and variables are there in java. Everything in java is object even the primitive data types can also be converted into object by using the wrapper class.

Robust

Java has the strong memory allocation and automatic garbage collection mechanism. It provides the powerful exception handling and type checking mechanism as compare to other programming languages. Compiler checks the program whether there any error and interpreter checks any run time error and makes the system secure from crash. All of the above features makes the java language robust.

Distributed

The widely used protocols like HTTP and FTP are developed in java. Internet programmers can call functions on these protocols and can get access the files from any remote machine on the internet rather than writing codes on their local system.

Portable

The feature Write-once-run-anywhere makes the java language portable provided that the system must have interpreter for the JVM. Java also have the standard data size irrespective of operating system or the processor. These features makes the java as a portable language.

Dynamic

While executing the java program the user can get the required files dynamically from a local drive or from a computer thousands of miles away from the user just by connecting with the Internet.

Secure

Java does not use memory pointers explicitly. All the programs in java are run under an area known as the sand box. Security manager determines the accessibility options of a class like reading and writing a file to the local disk. Java uses the public key encryption system to allow the java applications to transmit over the internet in the secure encrypted form. The bytecode Verifier checks the classes after loading.

Performance

Java uses native code usage, and lightweight process called threads. In the beginning interpretation of bytecode resulted the performance slow but the advance version of JVM uses the adaptive and just in time compilation technique that improves the performance.

Multithreaded

As we all know several features of Java like Secure, Robust, Portable, dynamic etc; you will be more delighted to know another feature of Java which is **Multithreaded.**

Java is also a Multithreaded programming language. Multithreading means a single program having different threads executing independently at the same time. Multiple threads execute instructions according to the program code in a process or a program. Multithreading works the similar way as multiple processes run on one computer.

Multithreading programming is a very interesting concept in Java. In multithreaded programs not even a single thread disturbs the execution of other thread. Threads are obtained from the pool of available ready to run threads and they run on the system CPUs. This is how Multithreading works in Java which you will soon come to know in details in later chapters.

Interpreted

We all know that Java is an interpreted language as well. With an interpreted language such as Java, programs run directly from the source code.

The interpreter program reads the source code and translates it on the fly into computations. Thus, Java as an interpreted language depends on an interpreter program.
The versatility of being **platform independent** makes Java to outshine from other languages. The source code to be written and distributed is platform independent.
Another advantage of Java as an interpreted language is its error debugging quality. Due to this any error occurring in the program gets traced. This is how it is different to work with Java.

Architecture Neutral

The term architectural neutral seems to be weird, but yes Java is an architectural neutral language as well. The growing popularity of networks makes developers think distributed. In the world of network it is essential that the applications must be able to migrate easily to different computer systems. Not only to computer systems but to a wide variety of hardware architecture and Operating system architectures as well. The Java compiler does this by generating byte code instructions, to be easily interpreted on any machine and to be easily translated into native machine code on the fly. The compiler generates an architecture-neutral object file format to enable a Java application to execute anywhere on the network and then the compiled code is executed on many processors, given the presence of the Java runtime system. Hence Java was designed to support applications on network. This feature of Java has thrived the programming language.

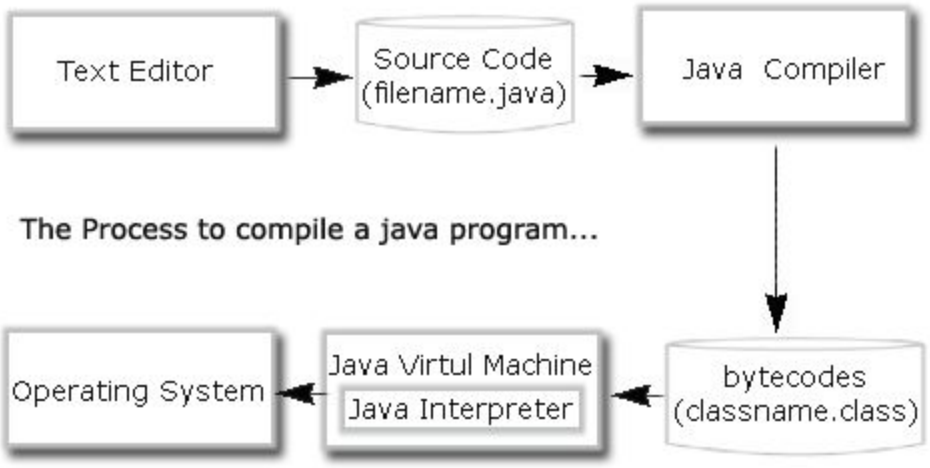
Java Compiler

To commence with Java programming, we must know the significance of Java Compiler. When we write any program in a text editor like Notepad, we use Java compiler to compile it. A Java Compiler **javac** is a computer program or set of programs which translates **java source code** into **java byte code.**

The output from a Java compiler comes in the form of **Java class files (with .class extension).** The java source code contained in files end with the **.java extension**. The file name must be the same as the class name, as **classname.java.** When the **javac** compiles the source file defined in a .java files, it generates bytecode for the java source file and saves in a class file with a .class extension.

The most commonly used Java compiler is javac, included in JDK from Sun Microsystems.

Following figure shows the working of the Java compiler:



Once the byte code is generated it can be run on any platform using Java Interpreter (JVM). It interprets byte code (.class file) and converts into machine specific **binary code**. Then JVM runs the binary code on the host machine.

How to use Java Compiler

When you run javac command on the command prompt, it shows the following output.

```
                                C:\>javac
                Usage: javac <options> <source files>
                       where possible options include:
           -g                         Generate all debugging info

           -g:none                    Generate no debugging info

         -g:{lines,vars,source}     Generate only some debugging info

              -nowarn                    Generate no warnings

 -verbose                   Output messages about what the compiler is doing

-deprecation                Output source locations where deprecated APIs are
                                                    used

    -classpath <path>          Specify where to find user class files and
                                            annotation processors

    -cp <path>                 Specify where to find user class files and
                                            annotation processors

     -sourcepath <path>         Specify where to find input source files

    -bootclasspath <path>      Override location of bootstrap class files

     -extdirs <dirs>            Override location of installed extensions

   -endorseddirs <dirs>       Override location of endorsed standards path
```

```
  -proc:{none,only}          Control whether annotation processing and/or
                                        compilation is done.

                -processor <class1>[,<class2>,<class3>...]
                                Names of the annotation processors to run;
                                    bypasses default discovery process

   -processorpath <path>      Specify where to find annotation processors

   -d <directory>             Specify where to place generated class files

   -s <directory>             Specify where to place generated source files

  -implicit:{none,class}     Specify whether or not to generate class files
                                       for implicitly referenced files

 -encoding <encoding>       Specify character encoding used by source files

   -source <release>          Provide source compatibility with specified
                                                release

   -target <release>          Generate class files for specific VM version

              -version                    Version information

      -help                      Print a synopsis of standard options

    -Akey[=value]              Options to pass to annotation processors

     -X                         Print a synopsis of nonstandard options

    -J<flag>                   Pass <flag> directly to the runtime system
                                    C:\>
```

Above output shows the different options of javac tool.

Using java compiler to compile java file:

Following example shows how a Compiler works. It compiles the program and gives the Syntax error, if there is any. Like in this example, we haven't initialized 'a' and we are using it in the next statement as 'int c=a+b'. That is why its showing a syntax error.

```
class A{
public static void
main(String[] args){
int a;
int b=2;
int c=a+b;

System.out.println(c);
}
}
```

Output of program:

```
C:\Program Files\Java\jdk1.6.0_01\bin>javac A.java
A.java:6: variable a might not have been initialized
int c=a+b;
^
1 error

C:\Program Files\Java\jdk1.6.0_01\bin>
```

Now, lets tweak this example. In this we have initialized 'a' as 'int a =2'. Hence, no syntax error has been detected.

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

Output of program:

```
C:\Program Files\Java\jdk1.6.0_01\bin>javac A.java

C:\Program Files\Java\jdk1.6.0_01\bin>java A
4
```

Java Interpreter

We can run Java on most platforms provided a platform must has a Java interpreter. That is why Java applications are platform independent. Java interpreter translates the **Java bytecode into the code that can be understood by the Operating System**. Basically, A Java interpreter is a software that implements the Java virtual machine and runs Java applications. As the Java compiler compiles the source code into the Java bytecode, the same way the Java interpreter translates the Java bytecode into the code that can be understood by the Operating System.

When a Java interpreter is installed on any platform that means it is **JVM** (Java virtual machine) enabled platform. It (Java Interpreter) performs all of the activities of the Java run-time system. It loads Java class files and interprets the compiled byte-code. You would be glad to know that some web browsers like Netscape and the Internet Explorer are Java

enabled. This means that these browsers contain Java interpreter. With the help of this Java interpreter we download the Applets from the Internet or an intranet to run within a web browser. The interpreter also serves as a specialized compiler in an implementation that supports dynamic or "just in time," compilation which turns Java byte-code into native machine instructions.

Throughout Java programming, we'll build both, **the standalone Java programs and applets.**

Sun's Java interpreter is called java. Lets learn how to start a standalone application with it. Load an initial class and specify it. Some options can also be specified to the interpreter, and any command-line arguments needed for the application as well:

% java [interpreter options] class name [program arguments]

The class should be specified as a fully qualified class name including the class package, if any.

Note : Moreover, we don't include the .class file extension. Here are a few examples:

% java animals.birds.BigBird
% java test

Once the class is loaded, java follows a C-like convention and searches for the class that contains a method called main(). If it finds an appropriate main() method, the interpreter starts the application by executing that method. From there, the application starts additional threads, reference other classes, and create its user interface.

Now, lets see how to go about an **Applet**. Although Java applet is a compiled Java code, the Java interpreter can't directly run them because they are used as part of a larger applications. For this we use **Java Applet Viewer**. It is a command line program to run Java applets. It is included in the SDK. It helps you to test an **applet** before you run it in a browser.

Data Types in Java

Java programming language is a language in which all the variables must be declared first and then to be used. That means to specify the name and the type of the variable. This specifies that Java is a strongly-typed programming language. Like

```
int pedal = 1;
```

This shows that there exists a field named 'pedal' that holds a data as a numerical value '1'. The values contained by the variables determines its data type and to perform the operations on it.

There are seven more primitive data types which are supported by Java language programming in addition to int. A primitive data type is a data type which is predefined in Java. Following are the eight primitive data types:

int

It is a 32-bit signed two's complement integer data type. It ranges from -2,147,483,648 to 2,147,483,647. This data type is used for integer values. However for wider range of values use long.

byte

The byte data type is an 8-bit signed two's complement integer. It ranges from -128 to127 (inclusive). We can save memory in large arrays using byte. We can also use byte instead of int to increase the limit of the code.

short

The short data type is a 16-bit signed two's complement integer. It ranges from -32,768 to 32,767. short is used to save memory in large arrays.

long

The long data type is a 64-bit signed two's complement integer. It ranges from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Use this data type with larger range of values.

float

The float data type is a single-precision 32-bits of storage . It ranges from 1.4e-45 to 3.4e+38 (positive or negative). Use a float (instead of double) to save memory in large arrays. We do not use this data type for the exact values such as currency.

Double

This data type is a double-precision 64-bits of storage . It ranges from 4.9e-324 to 1.8e +308 (positive or negative). This data type is generally the default choice for decimal values.

boolean

The boolean data type is 1-bit and has only two values: true and false. We use this data type for conditional statements. true and false are not the same as True and False. They are defined constants of the language.

char

The char data type is a single 16-bit, unsigned Unicode character. It ranges from 0 to 65,535. They are not same as ints, shorts etc.

The following table shows the default values for the data types: Keyword Description Size/Format

byte Byte-length integer 8-bit two's complement

short Short integer 16-bit two's complement

int Integer 32-bit two's complement

long Long integer 64-bit two's complement

float Single-precision floating point 32-bit

double Double-precision floating point 64-bit

char A single character 16-bit Unicode character

boolean A boolean value (true or false) true or false

When we declare a field it is not always essential that we initialize it too. The compiler sets a default value to the fields which are not initialized which might be zero or null. However this is not recommended.

Type Default Value

boolean false

byte 0

short 0

int 0

long 0L

char \u0000

float 0.0f

double 0.0d

Example :-Code of the Program :

public class identifierandpdatatype{

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

    byte byteident = 3;

    short shortident=100;

    int intident = 10;

    long longident = 40000;

    char charident = 'a';

    String stringident = "chandan";

    float floatident = 12.0045f;

    double doubleident = 2333333.000000000033343343434f;

    System.out.println(byteident + " is the value of identifier named 'byteident' which primitive data type is
byte.");

    System.out.println(shortident + " is the value of  identifier named 'shortident' which primitive data type
is short.");

    System.out.println(intident + " is the value of identifire named 'intident' which primitive data type is
int.");

    System.out.println(longident + " is the value of identifier named 'longident' which primitive data type is
long.");

    System.out.println(charident + " is the value of identifire named 'charident' which primitive data type is
char.");

    System.out.println(stringident + " is the value of identifire named 'stringident' which primitive data type
is string.");

    System.out.println(floatident + " is the value of identifier named 'floatident' which primitive data type
is float.");

    System.out.println(doubleident + " is the value  of identifier  named 'doubleident' which primitive data
type is double.");

  }

}
```

Understanding the hello world program

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

Type casting in Java

This example illustrates that what is *type casting*? **Type Casting** refers to changing an entity of one datatype into another. This is important for the type conversion in developing any application. If you will store a int value into a byte variable directly, this will be illegal operation. For storing your calculated int value in a byte variable you will have to change the type of resultant data which has to be stored. This type of operation has illustrated below :

In this example we will see that how to convert the data type by using type casting. In the given line of the code `c = (char)(t?1:0);` illustrates that if t which is `boolean type variable`

is true then value of c which is the char type variable will be 1 but 1 is a numeric value. So, 1 is changed into character according to the Unicode value. But in this line `c = (char)(t?'1':'0');` 1 is already given as a character which will be stored as it is in the char type variable c.

c

```
public class conversion{
  public static void main(String[] args){
    boolean t = true;
    byte b = 2;
    short s = 100;
    char c = 'C';
    int i = 200;
    long l = 24000;
    float f = 3.14f;
    double d = 0.000000000000053;
    String g = "string";
    System.out.println("Value of all the variables like");
    System.out.println("t = " + t );
    System.out.println("b = " + b );
    System.out.println("s = " + s );
    System.out.println("c = " + c );
    System.out.println("i = " + i );
    System.out.println("l = " + l );
    System.out.println("f = " + f );
    System.out.println("d = " + d );
    System.out.println("g = " + g );
    System.out.println();
    //Convert from boolean to byte.
    b = (byte)(t?1:0);
    System.out.println("Value of b after conversion : " + b);
    //Convert from boolean to short.
    s = (short)(t?1:0);
    System.out.println("Value of s after conversion : " + s);
    //Convert from boolean to int.
    i = (int)(t?1:0);
    System.out.println("Value of i after conversion : " + i);
    //Convert from boolean to char.
    c = (char)(t?'1':'0');
    System.out.println("Value of c after conversion : " + c);
    c = (char)(t?1:0);
    System.out.println("Value of c after conversion in unicode : " + c);
    //Convert from boolean to long.
    l = (long)(t?1:0);
    System.out.println("Value of l after conversion : " + l);
    //Convert from boolean to float.
    f = (float)(t?1:0);
    System.out.println("Value of f after conversion : " + f);
    //Convert from boolean to double.
    d = (double)(t?1:0);
    System.out.println("Value of d after conversion : " + d);
    //Convert from boolean to String.
    g = String.valueOf(t);
```

```
    System.out.println("Value of g after conversion : " + g);
    g = (String)(t?"1":"0");
    System.out.println("Value of g after conversion : " + g);
    int sum = (int)(b + i + l + d + f);
    System.out.println("Value of sum after conversion : " + sum);
  }
}
```

Operators

General Properties of Operators

In Java, an expression carries out some operation or operations that are directed by a list of allowed operators. An operator acts upon one, two or three operands. Here are some general properties of operators:

Operands

An operand can be:

- a numeric variable - integer, floating point or character
- any primitive type variable - numeric and boolean
- reference variable to an object
- a literal - numeric value, boolean value, or string.
- an array element, "a[2]"
- `char` primitive, which in numeric operations is treated as an unsigned two byte integer

The operator is *unary* if it acts on a single operand; *binary* if it requres two operands. The conditional operator is the only *ternary* operator in Java.

Each operator places specific requirements on the operand types allowed. For example, the subtractive operator "- " in `x=a-b;` requires that a and b variables be numeric types. The assignment operator "=" then requires that x also be a numeric type. (If a and b were wider types than x, a casting operation would also be required.)

Returned Value

A value is "returned" at the completion of an operation. The following statements use the assignment operator "=" and the addition operator "+"

```
int x = 3;
 int y = x+5;
```

and result in x holding the value 3 and y holding the value 8. The entire expression y= x+5 could be used in another expression:

```
int x=3;
 int y;
 int z = (y=x+5) * 4;
```

which results in y holding 8 and z holding 32. The assignment operator "=" in the expression "`y=x+5`" produces a new value for y and also *returns* the value of y to be used in the expression for z.

Effects on Operands

In most of the operations, the operands themselves are not changed. However, for some operators, the operand(s) do undergo a change:

- **Assignment operators**: "`x=y`" replaces the value of the first operand with that of the second. The other assignment operators, "`*=, +=`, etc" will also replace the value of the first operand but only after using its initial value in the particular operation indicated by the symbol before the = sign.

- **Increment & decrement operators**: The operand is incremented or decremented, before or after the operand's value is returned, depending on whether it is a pre- or post- increment of decrement.

If an operand is changed by the operation, note that if the statement holding that expression is processed again, e.g. in a loop, the resulting value can be different for each pass.

Expression Evaluation

The operands of an operator are always evaluated **left to right**. For example, in

```
x = a + b;
```

the first "+" operator will determine the value of `a` and then `b`.

Do not get this rule confused with the **precedence** and **associativity** rules.

- **Precedence** determines the order in which operators act in an expression of more than one operator. The table below gives the rating for each operator, the higher number indicating higher precedence.

- **Associativity** rules determine the grouping of operands and operators in an expression with more than one operator of the same precedence.

For example, in the expression

```
x = a + b * c;
```

the first "+" operator still first determines its left operand ("`a`" in this case) and then its right operand. But in this case the right operand consists of the expression "`b*c`". The multiplication operator "`*`" has a *higher precedence* than the additive "+".

Precedence can be overridden with parentheses, e.g.

```
x = (a + b) * c;
```

will force the addition of `b` to `a`, and then this sum is multiplied by `c`.

Although the precedence ratings, which are similar to those in C/C++, were chosen for the most "natural" ordering of the operator evaluations, it never hurts to use the parentheses if you are unsure of the precedence and don't have the table handy.

When the operations in an expression all have the same precedence rating, the *associativity* rules determine the order of the operations. For most operators, the evaluation is done left to right, e.g.

```
x = a + b - c;
```

Here, addition and subtraction have the same precedence rating and so `a` and `b` are added and then from this sum `c` is subtracted. Again, parentheses can be used to overrule the default associativity, e.g.

```
x = a + (b - c);
```

However, the assignment and unary operators, are associated right to left, e.g.

```
x += y -= -~4;
```

is equivalent to

```
x += (y -= (-(~4)));
```

or in long hand,

```
int a = ~4;
 a = -a;
 y = y - a;
 x = x + y;
```

Other Operator Tricks

Finally, here are some other miscellaneous notes about operators. As indicated in the last example above, assignment operations can be chained:

```
x = y = z = 4;
```

with the evaluations proceeding right to left.

Assignments can be combined into other operations, to compact the code, e.g.

```
if( (x = 5) == b) y = 10;
```

which first sets `x` to 5, then returns that value for the test of `b`. You should not overuse this technique else it makes the code unreadable. Occasionally, though, it can be a neat approach.

Tables of Java Operators

Assignment Operators

```
x operation= y
   is equivalent to
x = x operation y
```

x and y must be numeric or char types except for "=", which allows x and y also to be object references. In this case, x must be of the same type of class or interface as y. If mixed floating-point and integer types, the rules for mixed types in expressions apply.

`=`	Assignment operator. `x = y;` y is evaluated and x set to this value. The value of x is then returned.

`+=, -=, *=, /=, %=`	Arithmetic operation and then assignment, e.g.
	`x += y;`
	is equivalent to
	`x = x + y;`

`&=, |=, ^=`	Bitwise operation and then assignment, e.g.
	`x &= y;`
	is equivalent to
	`x = x & y;`

`<<=, >>=, >>>=`	Shift operations and then assignment, e.g.
	`x <<= n;`
	is equivalent to
	`x = x << n;`

Arithmetic Operators

x and y are numeric or char types. If mixed floating-point and integer types, then floating-point arithmetic used and a floating-point value returned. If mixed integer types, the wider type is returned. If double and float mixed, double is returned.

`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division If FP arithmetic and y = 0.0, then infinity returned if x is not zero, NaN if x is zero. ArthmeticException thrown if x & y are integer types and y is zero.

`x % y`	Modulo - remainder of x/y returned. If FP arithmetic and y = 0.0 or infinity, then NaN returned ArthmeticException thrown if x & y are integer types and y is zero.
`-x`	Unary minus Negation of x value

Increment & Decrement Operators x and y are numeric (FP & integer) or char types.	

`x++`	Post-increment : add 1 to the value.
	The value is returned *before* the increment is made, e.g.
	`x = 1;`
	`y = x++;`
	Then y will hold 1 and x will hold 2

`x--`	Post-decrement : subtract 1 from the value.
	The value is returned *before* the decrement is made, e.g. :
	`x = 1;`
	`y = x--;`
	Then y will hold 1 and x will hold 0.

`++x`	Pre-increment : add 1 to the value.
	The value is returned *after* the increment is made, e.g.
	`x = 1;`
	`y = ++x;`
	Then y will hold 2 and x will hold 2.

`--x`	Pre-decrement : subtract 1 from the value.
	The value is returned *after* the decrement is made, e.g.
	`x = 1;`
	`y = --x;`
	Then y will hold 0 and x will hold 0.

Boolean Operators x and y are boolean types. x and y can be expressions that result in a boolean value. Result is a boolean `true` or `false` value.		
`x && y`	Conditional AND	If both x and y are true, result is true. If either x or y are false, the result is false If x is false, y is not evaluated.

`x & y`	Boolean AND	If both x and y are true,the result is true. If either x or y are false, the result is false Both x and y are evaluated before the test.
`x || y`	Conditional OR	If either x or y are true, the result is true. If x is true, y is not evaluated.
`x | y`	Boolean OR	If either x or y are true, the result is true. Both x & y are evaluated before the test.
`!x`	Boolean NOT	If x is true, the result is false. If x is false, the result is true.
`x ^ y`	Boolean XOR	If x is true and y is false, the result is true. If x is false and y is true, the result is true. Otherwise, the result is false. Both x and y are evaluated before the test.

Comparison Operators

x and y are numeric or char types only except for "==" and "!=" operators, which can also compare references. If mixed types, then the narrower type converted to wider type. Returned value is boolean `true` or `false`.

`x < y`	Is x less than y ?
`x <= y`	Is x less than or equal to y ?
`x > y`	Is x greater than y ?
`x >= y`	Is x greater than or equal to y ?
`x == y`	Is x equal to y ?
`x != y`	Is x not equal to y ?

Bitwise Operators

x and y are integers. If mixed integer types, such as `int` and `long`, the result will be of the wider type.

Note: Operations on `byte` and `short` types may give unexpected results since operands are promoted to integers during intermediate operations. For example,

```
byte x = (byte)0xFF;
x >>>= 1;
```

will result in `0xFF` in x rather than `0x7F`. That is because the operation is carried out on a signed integer rather than simply on 8 bits. Here the signed byte is promoted to the signed integer 0xFFFFFFFF.

Use of an integer would go as follows:

```
int i = 0xFF;
i >>>= 1;
```

This results in `0x7F` in the variable i.

`~x`	Compliment	Flip each bit, ones to zeros, zeros to ones
`x & y`	AND	AND each bit a with corresponding bit in b
`x | y`	OR	OR each bit in a with corresponding bit in b
`x ^ y`	XOR	XOR each bit in x with corresponding bit in y
`x << y`	Shift left	Shift x to the left by y bits. High order bits lost. Zero bits fill in right bits.
`x >> y`	Shift Right - Signed	Shift x to the right by y bits. Low order bits lost. Same bit value as sign (0 for positive numbers, 1 for negative) fills in the left bits.
`x >>> y`	Shift Right - Unsigned	Shift x to the right by y bits. Low order bits lost. Zeros fill in left bits regardless of sign.

Class and Object Operators		
`x instanceof c`	Class Test Operator	The first operand must be an object reference. *c* is the name of a class or interface. If x is an instance of type c or a subclass of c, then `true` returned. If x is an instance of interface type c or a sub-interface, then `true` is returned. Otherwise, `false` is returned.
`new c(args)`	Class Instantiation	Create an instance of class c using constructor `c(args)`
`"."`	Class Member Access	Access a method or field of a class or object : `o.f` - field access for object o `o.m()` - method access for object o

`()`	Method Invocation	Parentheses after a method name invokes (i.e. calls) the code for the method, e.g.
		`o.m()`
		`o.m(x,y)`

`(c)`	Object Cast	Treat an object as the type of class or interface c: `c x=(c)y;` Treat y as an instance of class or interface c

`+`	String Concatenation	This binary operator will concatenate one string to another. E.g.
		`String str1 = "abc";`
		`String str2 = "def";`
		`String str3 = str1 + str2`
		results in `str3` holding `"abcdef"`.
		For mixed operands, if either a or b in (a + b) is a string, concatenation to a string will occur. Primitives will be converted to strings and the toString() methods of objects will be called.
		(This is the only case of operator *overloading* in Java.)
		Note that the equivalence operator "+=" will also perform string concatenation.

`[]`	Array Element Access	In Java, arrays are classes. However, the bracket operators work essentially the same as in the C/C++.
		To access a given element of an array, place the number of the element as an `int` value (`long` values cannot be used in Java arrays) into the brackets, e.g.
		`float a = b[3];`
		`int n = 5;`
		`char c=c[n];`
		where `b` is a `float` array and `c` is a char array.

Other Operators		
`x=`*`boolean`*`?y:x`	Conditional Operator	The first operand - *boolean* - is a boolean variable or expression. First this boolean operand is evaluated. If it is `true` then the second operator evaluated and x is set to that value. If the boolean operator is `false`, then the third operand is evaluated and x is set to that value.

(*primitive type*)	Type Cast	To assign a value of one primitive numeric type a more narrow type, e.g. long to int, an explicit cast operation is required, e.g.
		`long a = 5;`
		`int b = (int)a;`

Operator Precedence

The larger the number, the higher the precedence.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
= *= /= %= += -= <<= >>= >>>= &= ^= |=	?:	||	&&	|	^	&	== !=	< <= > >=	<< >> >>>	+ -	* / %	new (*type*)	++x --x +x -x ~ !	. [] (args) x++ x--

Notes:

- `(type)` refers to the casting operator
- "." is the object member access operator
- `[]` is the array access operator
- `(args)` indicates the invocation of a method.
- In column 11, the + and - refer to binary addition and subtraction. Also, the + refers tothe string append operator. Whereas in column 14, the + & - refer to the unary operations `+x` and `-x` specify the sign of the value.
- |, ^, and & refer to both the bitwise and boolean operators.

Operator Associativity

The following operators have **Right to Left** associativity. All other operators (see precedence table above) are evaluated left to right.

= *= /= %= += -= <<= >>= >>>= &= ^= |=	?: new (*type* cast) ++x --x +x -x ~ !

Literals

When you explicitly assign a value to a variable in the code, as in

```
double x = 3.1;
 int i = 43;
```

the compiler needs to translate the character strings "3.1" and "43" into numerical values of a particular type. Such explicit values in a computer language are called "literals" (for the obvious reason that they are *literally* equivalent to their stated value.)

(In the bytecode each literal in your class code gets put into a so-called *constants pool*.)

Floating Point Literals

The compiler will see the decimal point and decide that the string "3.1" is a valid floating point string. It must next decide what type to assign it. In Java a floating point literal **defaults** to the `double` type.

```
double x = 3.1;
```

Since converting a double to float is a narrowing conversion, the following assignment

```
float y = 3.1; // results in an error message
```

causes an error. You must instead explicitly type the floating point literal,

```
float y = 3.1f;
```

by appending the "f" character to the value.

The following examples illustrate how to express floating point exponents in a literal:

```
double y = 3.1e10;
 y = 3.1e-10;
 y = 3E10;
 y = -3.1e05;
```

Integer Literals

Integer literals default to the `int` type.

```
int i = 3;   // 3 is an integer literal
```

You can assign a `long` type to the `int` literal since this will result in a widening conversion, which does not require an explicit cast.

```
long m = 3; // allowed
```

You might think, then, that the following narrowing conversions would result in an error

```
byte b = -2;
 short i = 3;
```

but Java allows this if the literals are within the allowed ranges for `byte` and `short` types. However, this only works with the direct literal assignment. An assignment to an `int` variable,

```
int i = 3;
 short i = i; // error
 i = -2;
 byte b = i; // error
```

will result in an error message.

You can explicitly specify a long literal by appending a 'L' or 'l':

```
long m = 2147483648L;
```

Besides the decimal base, you can use hexadecimal and octal formats for the integer literals:

```
short i = 0x00AF; // hex for 175
 byte b = 017;     // octal for 15
```

where hex numbers begin with "0x" and octal with "0".

Note: Assigning a literal value larger than the range for a particular integer type will *NOT* result in an overflow warning (and no underflow warning for large negative values.) Instead the value will *wrap around* to the lowest value. For example,

```
int i = 130;
 byte bb = (byte)i;
```

results in a value of -126 in the bb variable.

Characters and String Literals

You must bracket a character literal with *single* quotation marks.

```
char c = '*';
 char c1 = 'A';
 char c2 = "A"; // error
```

A string literal, however, requires *double* quotation marks:

```
String str = "abc";
 String str1 = "Java";
 String str2 = 'J'; // error
```

Special Literals

For boolean type data, only two types of data are available: `true` and `false`. So these literals are defined already in the core language:

```
Boolean b = true;
 Boolean b1 = false;
```

Similarly, for references to objects, the `null` literal can be used when no object is yet chosen for the reference variable:

```
String str = null;
```

Operators

An *expression* produces a result and returns a value. Examples include:

- `i = 2` : the assignment puts 2 into the i variable and returns the value 2
- `k++` : returns k, then k is incremented by 1
- `x < y` : logical "less than" comparison, returns a Boolean true or false value
- `i | j` : returns the value of a bitwise OR operation on bits in the two variables.
- `4.0*Math.sin(i * Math.PI)` : combines several operations in this expression including multiplication & a method call.

Expressions involve at least one operator. A single operator can have 1, 2 or 3 operands.

Effects on Operands

In most of the operations, the operands themselves are not changed. However, for some operators, the operand(s) do undergo a change:

Assignment operators

"`x = y`" replaces the value of the first operand with that of the second.
The other assignment operators, "`*=, +=, -=, /=`" also replace the value of the first operand but only after using its initial value in the operation indicated by the symbol before the equals sign.

For example,

```
x *= y
```

results in x replaced by `x * y`. Also, this is the value returned from the operation.

Increment and decrement operators:

`(++x)` x is incremented before its value is returned.
`(--x)` x is decremented before its value is returned.
`(x++)` the initial value of x is returned and then x is incremented.
`(x--)` the initial value of x is returned and then x is decremented.

For the increment and decrement operations, note that in a standalone expression such as

```
x++;
```

there is no effective difference between `x++` and `++x`. Both expressions increment the value stored in the variable x. However, in expressions such as

```
y = x++;
```

and

```
z = ++i;
```

the order of the appearance of the increment operator is important. In the former case, y takes on the value of x before the increment occurs. If x is initially 3, then y becomes 3 and x becomes 4. In the latter case the increment occurs before the value is used. So an initial value of 3 for i leads to i incrementing to 4 and then z taking on the new value, 4.

Remember that if an operand is changed by the operation and the statement holding that expression is processed again, as in a loop, the operand's value will be different for each pass.

Returned Value

A value is returned from the expression. The following statements use the assignment operator "=" and the addition operator "+"

```
int x = 3;
 int y = x + 5;
```

These statements result in x holding the value 3 and y holding the value 8. The entire expression `y = x + 5` could be used in another expression:

```
int x = 3;
 int y;
 int z = (y = x + 5) * 4;
```

This results in y holding 8 and z holding 32. The assignment operator "=" in the expression

```
(y = x + 5)
```

produces a new value for y and also returns the value of y to be used in the expression for z.

Expression Evaluation

The operands of an operator are always evaluated left to right. For example, in

```
x = a + b;
```

the "+" operator will determine the value of expression a and then expression b. Do not get this rule confused with the precedence and associativity rules, discussed next.

Precedence determines the order in which operators act in an expression with more than one operator. The Operator Precedence Table gives the precedence rating for each operator, the higher number indicating higher precedence.

Associativity rules determines how the compiler groups the operands and operators in an expression with more than one operator of the same precedence. For example, in the expression

```
x = a + b * c;
```

the evaluation begins with a and then the "+" operator determines its right operand. But in this case the right operand consists of the expression "b * c". The multiplication operator "*" has a higher precedence than the additive operator "+" so b multiplies c rather than sums with a. Precedence can be overridden with parentheses, as in

```
x = (a + b) * c;
```

The parentheses force the addition of b to a, and then c multiplies this sum.

Although the precedence ratings, which are similar to those in C/C++, were chosen for the most "natural" ordering of the operator evaluations, it never hurts to use the parentheses if you are unsure of the precedence and to make the code more readable.

When the operations in an expression all have the same precedence rating, the associativity rules determine the order of the operations. For most operators, the evaluation is done left to right, as in

```
x = a - b + c;
```

Here, addition and subtraction have the same precedence rating and so a and b are subtracted and then c added to the difference. Again, parentheses can be used to overrule the default associativity, as in

```
x = a - (b + c);
```

The assignment and unary operators, on the other hand, are associated right to left. For example, the statement

```
x += y -= -~4;
```

is equivalent to

```
x += (y -= (-(~4)));
```

or, in long hand,

```
int a = ~4;
 a = -a;
 y = y - a;
 x = x + y;
```

Casting

Converting one type of data into another must follow the rules of *casting*. If a conversion results in the loss of precision, as in an `int` value converted to a `short`, then the compiler will issue an error message unless an explicit cast is made.

To convert type `AA` data into type `BB` data, put the type `BB` name in parentheses in front of the type `AA` data:

```
AA a = aData;
BB b = (BB)a; // cast type AA to type BB
```

For example, to convert integer data to floating point:

```
int i=0;
float f;
f=(float)i; // Cast int as float
```

Expressions can promote to a wider type without an explicit cast:

```
int i=1;
long j=3L; // Literals are int types so require L suffix
j=i;       // OK
```

However, you can not assign a value to a more narrow type without an explicit cast**:**

```
i=j;       // Error in assigning long to int
i=(int)j;  // OK
```

So a data type with lower precision (fewer bits) can be converted to a type of higher precision without explicit casting. To convert a higher precision type to a lower precision, however, an explicit cast is required or the compiler will flag an error.

Note that when you cast a value of a wider type down to a more narrow type, such as an `int` value to a `byte` variable, the upper bytes will be truncated. That is, the lowest order byte in the `int` value will be copied to the `byte` value.

Primitive Type Conversion Table

Below is a table that indicates to which of the other primitive types you can cast a given primitive data type. The symbol **C** indicates that an explicit cast is required since the precision is decreasing. The symbol **A** indicates that the precision is increasing so an automatic cast occurs without the need for an explicit cast. **N** indicates that the conversion is not allowed.

	int	**long**	**float**	**double**	**char**	**byte**	**short**	**boolean**
int	-	A	A*	A	C	C	C	N
long	C	-	A*	A*	C	C	C	N
float	C	C	-	A	C	C	C	N
double	C	C	C	-	C	C	C	N
char	A	A	A	A	-	C	C	N
byte	A	A	A	A	C	-	A	N
short	A	A	A	A	C	C	-	N
boolean	N	N	N	N	N	N	N	-

The * asterisk indicates that the least significant digits may be lost in the conversion even though the target type allows for bigger numbers. For example, a large value in an `int` type value that

uses all 32 bits will lose some of the lower bits when converted to `float` since the exponent uses 8 bits of the 32 provided for `float` values.

Mixed Types in an Expression

If an expression holds a **mix** of types, the lower precision or narrower value operand is converted to a higher precision or wider type. This result then must be cast if it goes to a lower precision type:

```
float x,y=3;
int j,i=3;
x= i*y;       // OK since i will be promoted to float
j= i*y;       // Error since result is a float value
j= (int)(i*y) // OK
```

The process of converting a value to a wider or higher precision integer or floating point type is called "numeric promotion". The Java VM specification states the following rules for promotion in an expression of two operands, as in `x+i`:

- If either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.

The java programming language supports the following types of controlling statements such as:

1.The break statement
2.The continue statement
3.The return statement

Continue: The continue statement is used in many programming languages such as C, C++, java etc. Sometimes we do not need to execute some statements under the loop then we use the continue statement that stops the normal flow of the control and control returns to the loop without executing the statements written after the continue statement. There is the difference between break and continue statement that the break statement exit control from the loop but continue statement keeps continuity in loop without executing the statement written after the continue statement according to the conditions.

In this program we will see that how the continue statement is used to stop the execution after that.

Here is the code of the program :

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

```
    Thread t = new Thread();
    int a = 0;
    try{
      for (int i=1;i<10;i++)
      {
        if (i == 5)
        {
          continue;
          //control will never reach here (after the continue statement).
          //a = i;
        }
        t.sleep(1000);
        System.out.println("chandan");
        System.out.println("Value of a : " + a);
      }
    }
    catch(InterruptedException e){}
  }
}
```

Output of the program :

If we write the code in the given program like this :

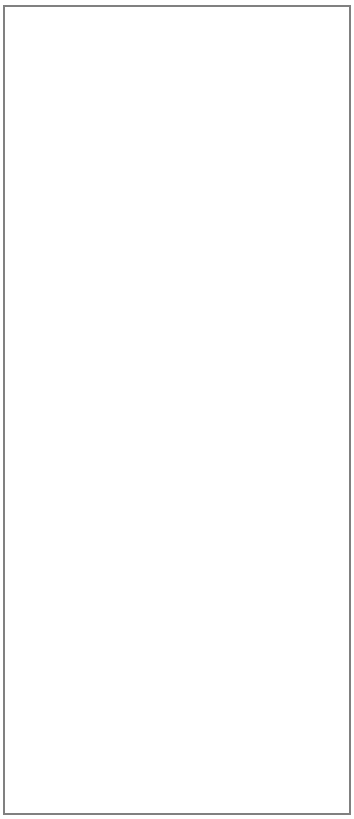
```
if (i == 5 )
{
   continue;
   a = i;
}
```

Then the program will generate a error on compile time like :

If we write the code in the given program like this :

```
if (i == 5 )
{
   continue;
}
```

Then the program prints the output like :



Break in Java

Break: The break statement is used in many programming languages such as c, c++, java etc. Some times we need to exit from a loop before the completion of the loop then we use break statement and exit from the loop and loop is terminated. The break statement is used in while loop, do - while loop, for loop and also used in the switch statement.

Code of the program :

```
public class  Break{
  public static void main(String[] args){
    int i,j;
    System.out.println("Prime numbers between 1 to 50 : ");
    for (i = 1;i < 50;i++ ){
      for (j = 2;j < i;j++ ){
        if(i % j == 0)
        {
          break;
        }
      }
      if(i == j)
```

```
      {
        System.out.print("  " + i);
      }
    }
  }
}
```

Output of the program :

The return Statement

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The return statement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

```
return;
```

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then and if-then-else Statements

The if-then Statement

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the applyBrakes method could be as follows:

```
void applyBrakes(){
    if (isMoving){  // the "if" clause: bicycle must be moving
        currentSpeed--; // the "then" clause: decrease current speed
    }
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes(){
    if (isMoving) currentSpeed--; // same as above, but without
braces
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the applyBrakes method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes(){
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
```

```
}
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

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

        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

The output from the program is:

```
Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

The switch Statement

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* (discussed in Classes and Inheritance) and a few special classes that "wrap" certain primitive types: Character, Byte, Short, and Integer (discussed in Simple Data Objects).

The following program, SwitchDemo, declares an int named month whose value represents a month out of the year. The program displays the name of the

month, based on the value of month, using the switch statement.

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

        int month = 8;
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Invalid month.");break;
        }
    }
}
```

In this case, "August" is printed to standard output.

The body of a switch statement is known as a *switch block*. Any statement immediately contained by the switch block may be labeled with one or more case or default labels. The switch statement evaluates its expression and executes the appropriate case.

Of course, you could also implement the same thing with if-then-else statements:

```
int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
. . . // and so on
```

Deciding whether to use if-then-else statements or a switch statement is sometimes a judgment call. You can decide which one to use based on readability and other factors. An if-then-else statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer or enumerated value.

Another point of interest is the break statement after each case. Each break

statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, case statements fall through; that is, without an explicit break, control will flow sequentially through subsequent case statements. The following program, SwitchDemo2, illustrates why it might be useful to have case statements fall through:

```
class SwitchDemo2 {
   public static void main(String[] args) {

      int month = 2;
      int year = 2000;
      int numDays = 0;

      switch (month) {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            numDays = 31;
            break;
         case 4:
         case 6:
         case 9:
         case 11:
            numDays = 30;
            break;
         case 2:
            if ( ((year % 4 == 0) && !(year % 100 == 0))
                 || (year % 400 == 0) )
               numDays = 29;
            else
               numDays = 28;
            break;
         default:
            System.out.println("Invalid month.");
            break;
      }
      System.out.println("Number of Days = " + numDays);
   }
}
```

This is the output from the program.

```
Number of Days = 29
```

Technically, the final break is not required because flow would fall out of the switch statement anyway. However, we recommend using a break so that

modifying the code is easier and less error-prone. The default section handles all values that aren't explicitly handled by one of the case sections.

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {
    statement(s)
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){
   // your code goes here
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
```

```
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count <= 11);
    }
}
```

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (initialization; termination; increment) {
   statement(s)
}
```

When using this version of the for statement, keep in mind that:

- The *initialization* expression initializes the loop; it's executed once, as the loop begins.
- When the *termination* expression evaluates to false, the loop terminates.
- The *increment* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

The output of this program is:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
```

```
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
for ( ; ; ) {    // infinite loop

    // your code goes here
}
```

The for statement also has another form designed for iteration through Collections and arrays This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
          System.out.println("Count is: " + item);
        }
    }
}
```

In this example, the variable item holds the current value from the numbers array. The output from this program is the same as before:

```
Count is: 1
Count is: 2
```

```
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

We recommend using this form of the for statement instead of the general form whenever possible.

Summary of Control Flow Statements

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. The while and do-while statements continually execute a block of statements while a particular condition is true. The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once. The for statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

Variables

As you learned in the previous lesson, an object stores its state in *fields*.

```
int cadence = 0;
    int speed = 0;
    int gear = 1;
```

The What Is an Object? discussion introduced you to fields, but you probably have still a few questions, such as: What are the rules and conventions for naming a field? Besides int, what other data types are there? Do fields have to be initialized when they are declared? Are fields assigned a default value if they are not explicitly initialized? We'll explore the answers to such questions in this lesson, but before we do, there are a few technical distinctions you must first become aware of. In the Java programming language, the terms "field" and "variable" are both used; this is a common source of confusion among new developers, since both often seem to refer to the same thing.

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)** Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the `static` keyword. Non-static fields are also known as *instance variables* because their values are unique to each *instance* of a class (to each object, in other words); the `currentSpeed` of one bicycle is independent from the `currentSpeed` of another.

- **Class Variables (Static Fields)** A *class variable* is any field declared with the `static` modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as `static` since conceptually the same number of gears will apply to all instances. The code `static int numGears = 6;` would create such a static field. Additionally, the keyword `final` could be added to indicate that the number of gears will never change.

- **Local Variables** Similar to how an object stores its state in fields, a method will often store its temporary state in *local variables*. The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

- **Parameters** You've already seen examples of parameters, both in the `Bicycle` class and in the `main` method of the "Hello World!" application. Recall that the signature for the `main` method is `public static void main(String[] args)`. Here, the `args` variable is the parameter to this method. The important thing to remember is that parameters are always classified as "variables" not "fields". This applies to other parameter-accepting constructs as well (such as constructors and exception handlers) that you'll learn about later in the tutorial.

Having said that, the remainder of this tutorial uses the following general guidelines when discussing fields and variables. If we are talking about "fields in general" (excluding local variables and parameters), we may simply say "fields". If the discussion applies to "all of the above", we may simply say "variables". If the context calls for a distinction, we will use specific terms (static field, local variables, etc.) as appropriate. You may also occasionally

see the term "member" used as well. A type's fields, methods, and nested types are collectively called its *members*.

Naming

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- Variable names are case-sensitive. A variable's name can be any legal identifier — an unlimited-length sequence of Unicode letters and digits, beginning with a letter, the dollar sign "$", or the underscore character "_". The convention, however, is to always begin your variable names with a letter, not "$" or "_". Additionally, the dollar sign character, by convention, is never used at all. You may find some situations where auto-generated names will contain the dollar sign, but your variable names should always avoid using it. A similar convention exists for the underscore character; while it's technically legal to begin your variable's name with "_", this practice is discouraged. White space is not permitted.

- Subsequent characters may be letters, digits, dollar signs, or underscore characters. Conventions (and common sense) apply to this rule as well. When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting; fields named `cadence`, `speed`, and `gear`, for example, are much more intuitive than abbreviated versions, such as `s`, `c`, and `g`. Also keep in mind that the name you choose must not be a keyword or reserved word.

- If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word. The names `gearRatio` and `currentGear` are prime examples of this convention. If your variable stores a constant value, such as `static final int NUM_GEARS = 6`, the convention changes slightly, capitalizing every letter and separating subsequent words with the underscore character. By convention, the underscore character is never used elsewhere.

Arrays

An *array* is a container object that holds a fixed number of values of a single type.

The length of an array is established when the array is created. After creation, its length is fixed. You've seen an example of arrays already, in the `main` method of the "Hello World!" application. This section discusses arrays in greater detail.

An array of ten elements

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the above illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, ArrayDemo, creates an array of integers, puts some values in it, and prints each value to standard output.

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray;            // declares an array of integers

        anArray = new int[10];      // allocates memory for 10 integers

        anArray[0] = 100; // initialize first element
        anArray[1] = 200; // initialize second element
        anArray[2] = 300; // etc.
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
```

```
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you'd probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as shown above. However, this example clearly illustrates the array syntax. You'll learn about the various looping constructs (`for`, `while`, and `do-while`) in the Control Flow section.

Declaring a Variable to Refer to an Array

The above program declares `anArray` with the following line of code:

```
int[] anArray;        // declares an array of integers
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as *type*`[]`, where *type* is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the naming section. As with variables of other types, the declaration does not actually create an array — it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

You can also place the square brackets after the array's name:

```
float anArrayOfFloats[]; // this form is discouraged
```

However, convention discourages this form; the brackets identify the array type

and should appear with the type designation.

Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
anArray = new int[10];  // create an array of integers
```

If this statement were missing, the compiler would print an error like the following, and compilation would fail:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900,
1000};
```

Here the length of the array is determined by the number of values provided between *{* and *}*.

You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

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

```
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                            {"Smith", "Jones"}};
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones
    }
}
```

The output from this program is:

```
Mr. Smith
Ms. Jones
```

Finally, you can use the built-in `length` property to determine the size of any array. The code

```
System.out.println(anArray.length);
```

will print the array's size to standard output.

Copying Arrays

The `System` class has an `arraycopy` method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

The two `Object` arguments specify the array to copy *from* and the array to copy *to*. The three `int` arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, ArrayCopyDemo, declares an array of `char` elements, spelling the word "decaffeinated". It uses `arraycopy` to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
```

```
}
```

The output from this program is:

```
caffein
```