JAVA FOUNDATION

TABLE OF CONTENTS

JAVA INTRODUCTION

Java is a programming language and computing platform first released by Sun Microsystems in 1995.
Java is fast, secure, and reliable.
From laptops to datacenters, game consoles to scientific supercomputers, cell phones to the Internet, Java is everywhere!
Java is Platform independent

SETTING UP THE ENVIRONMENT

Windows 10 and Windows 8

1. Click Windows Start Menu
2. In Search, search for “Edit environment variable for your account” then click an open it.
3. Environment variable window – Click New button
4. Enter Variable name: JAVA_HOME
5. Enter Variable value: C:\Program Files\Java\jdk1.8.0_25\
6. Click OK button
7. Click New button
8. Enter Variable name: Path
9. Enter Variable value: C:\Program Files\Java\jdk1.8.0_25\bin
10. Click OK

VERIFY JAVA ENVIRONMENT

Windows 10 and Windows 8

1. Click Windows Start Menu
2. In Search, search for “**cmd**”
3. When prompt appears type “**java -version**” in command line and enter
4. Following output should be displayed

```
java version "1.8.0_25"
Java(TM) SE Runtime Environment (build 1.8.0_25-b18)
Java HotSpot(TM) 64-Bit Server VM (build 25.25-b02, mixed mode)
```

DIFFERENCES BETWEEN JDK, JRE AND JVM

JDK

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

JRE

JRE stands for **“Java Runtime Environment”** and may also be written as **“Java RTE.”**The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the *Java Virtual Machine (JVM), core classes*, and *supporting files*.

JVM

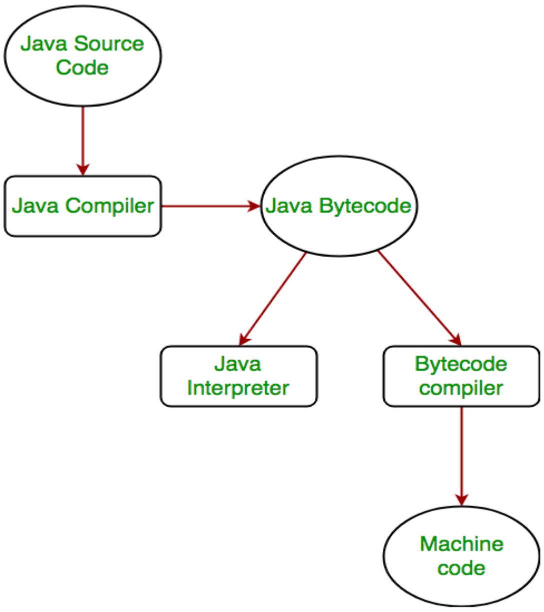
(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

JAVA PLATFORM INDEPENDENT

The meaning of platform independent is that, the java source code can run on all operating systems. For the source code to be understood by the machine, it needs to be in a language understood by machines, typically a machine-level language. So, here comes the role of a compiler. The compiler converts the high-level language (human language) into a format understood by the machines.

Step by step Execution of Java Program:

- Whenever, a program is written in JAVA, the javac compiles it.
- The result of the JAVA compiler is the **.class file or the bytecode** and not the machine native code.
- The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.
- And finally program runs to give the desired output.



JAVA CLASS FILE

A **Java class file** is a file containing Java bytecode and having **.class extension** that can be executed by JVM. A Java class file is created by a Java compiler from *.java*files as a result of successful compilation. As we know that a single Java programming language source file (*or we can say .java file*) may contain one class or more than one class. So if a *.java* file has more than one class then each class will compile into a separate class files.

For Example: Save this below code as **Test.java** on your system.

```
package sample;
public class Helloworld {
// Compiling this Java program would
// result in multiple class files.
    class Sample
    {

    }

    // Class Declaration
    class Student
    {

    }
    // Class Declaration
    static class Test
    {
        public static void main(String[] args)
        {
            System.out.println("Class File Structure");
        }
    }
}
```

FOR COMPILING:

```
javac Test.java
```

After compilation there will be **4 class** files in corresponding folder named as:

- Helloworld$Sample
- Helloworld$StudentTest.class
- Helloworld$Test
- Helloworld

JAVA NAMING CONVENTIONS

Naming conventions of java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

CAMEL CASE IN JAVA PROGRAMMING: It consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital.

CLASSES AND INTERFACES:

Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalised. Interfaces name should also be capitalised just like class names.
Use whole words and must avoid acronyms and abbreviations.

Examples:

```
interface  Bicycle
class MountainBike implements Bicyle

interface Sport
class Football implements Sport
```

METHODS:

Methods should be **verbs**, in mixed case with the **first letter lowercase**and with the first letter of each internal word capitalised.

Examples:

```
void changeGear(int newValue);
void speedUp(int increment);
void applyBrakes(int decrement);
```

VARIABLES:

- Variable names should be short yet meaningful.
- Should **not** start with underscore ('_') or dollar sign '$' characters.
- **One-character variable names should be avoided** except for temporary variables.
- Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Examples:

```
// variables for MountainBike class
int speed = 0;
int gear = 1;
```

CONSTANT VARIABLES:

- Should be **all uppercase** with words separated by underscores ("_").
- There are various constants used in predefined classes like Float, Long, String etc.

Examples:

```
static final int MIN_WIDTH = 4;
```

PACKAGES:

- The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
- Subsequent components of the package name vary according to an organisation's own internal naming conventions.

Examples:

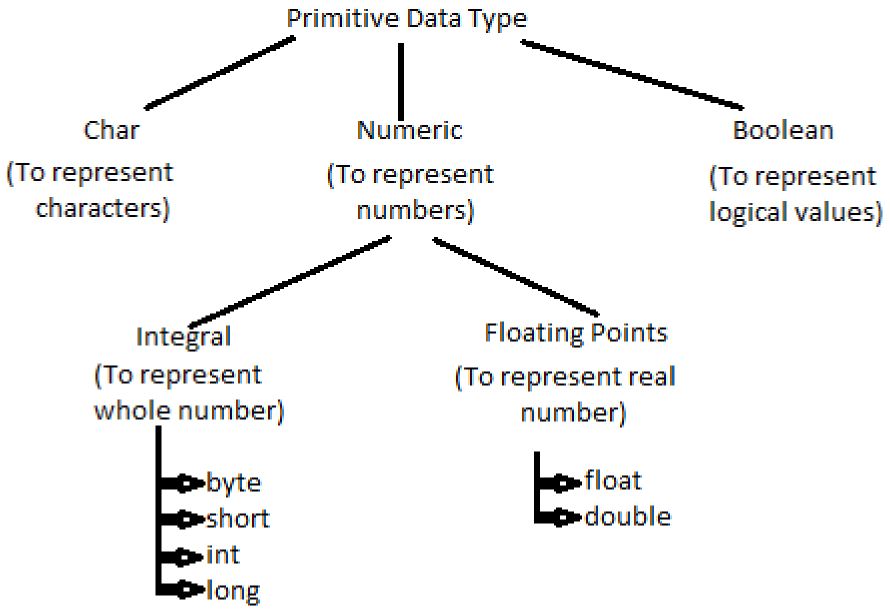
```
com.sun.eng
com.apple.quicktime.v2
```

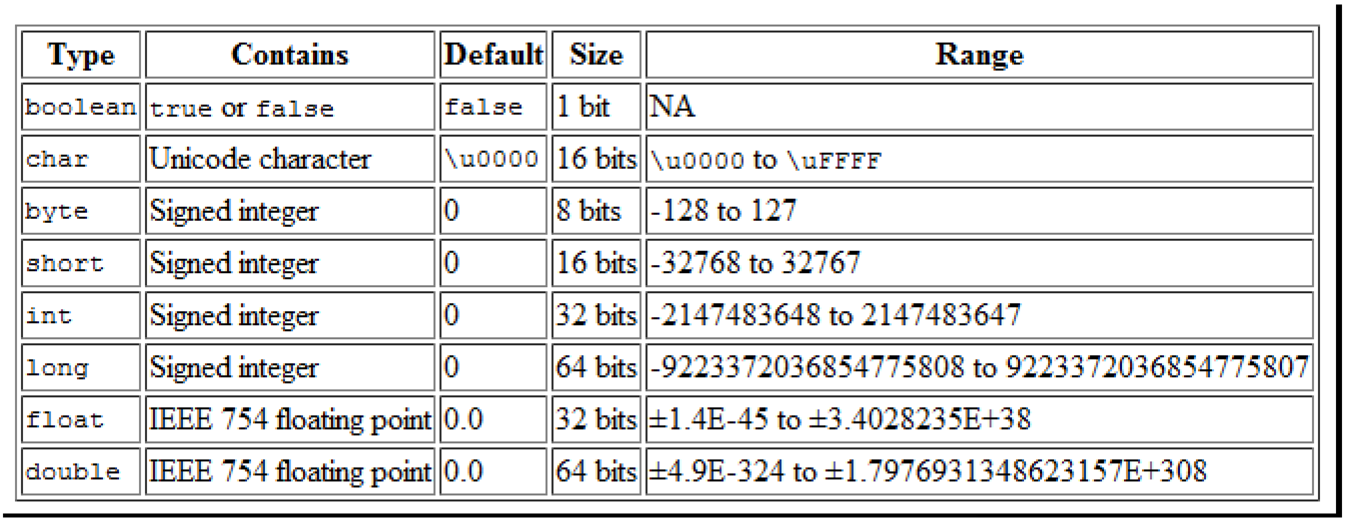
PRIMITIVE DATA TYPES

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.

```
int gear = 1;
```

Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:





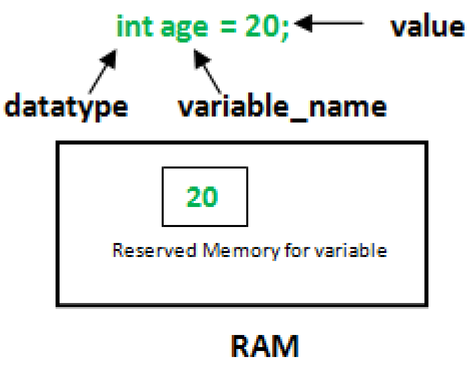
Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	IEEE 754 floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	IEEE 754 floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

A variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

HOW TO DECLARE VARIABLES?

We can declare variables in java as follows:



datatype: Type of data that can be stored in this variable.
variable_name: Name given to the variable.
Value: It is the initial value stored in the variable.

Examples:

```
float simpleInterest; //Declaring float variable
int time = 10, speed = 20; //Declaring and Initializing integer variable
char var = 'h'; // Declaring and Initializing character variable
```

TYPES OF VARIABLES

There are three types of variables in Java:

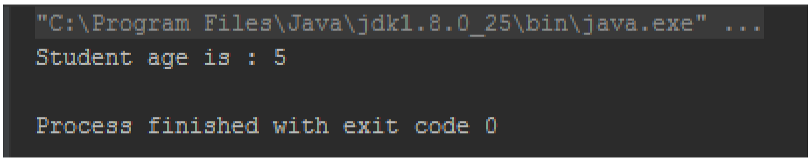
- Local Variables
- Instance Variables
- Static Variables

Local Variables: A variable defined within a block or method or constructor is called local variable.

Sample Program

```
package sample;
public class StudentDetails {
    public void StudentAge()
    {
        // local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
    }

    public static void main(String args[])
    {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

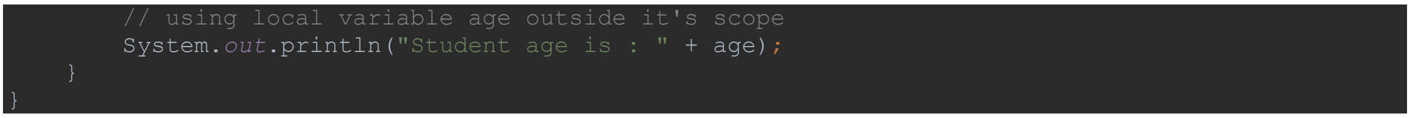
Output:



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Student age is : 5

Process finished with exit code 0
```

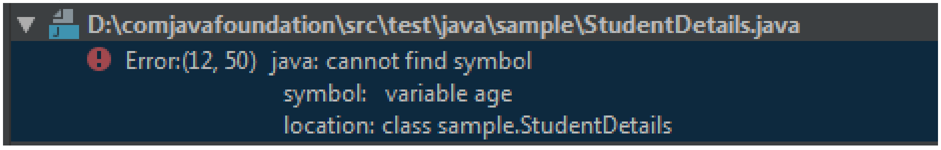
Sample Program

```
package sample;
public class StudentDetails {
    public void StudentAge()
    { // local variable age
        int age = 0;
        age = age + 5;
    }

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



```
        // using local variable age outside it's scope
        System.out.println("Student age is : " + age);
    }
}
```

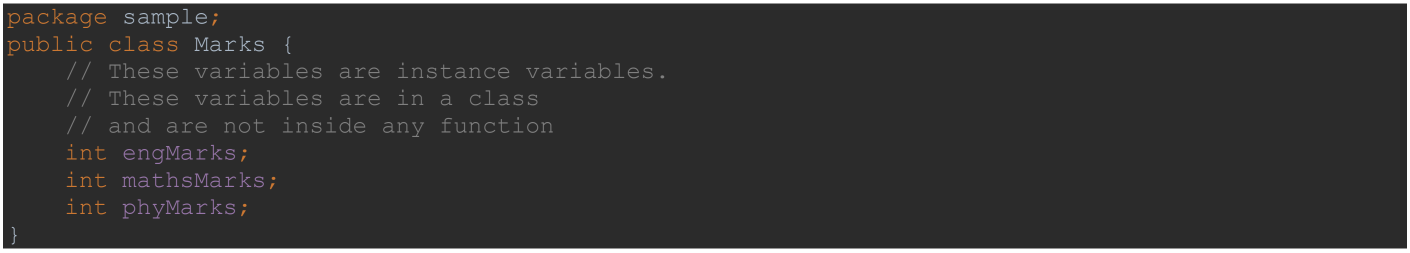
Output:



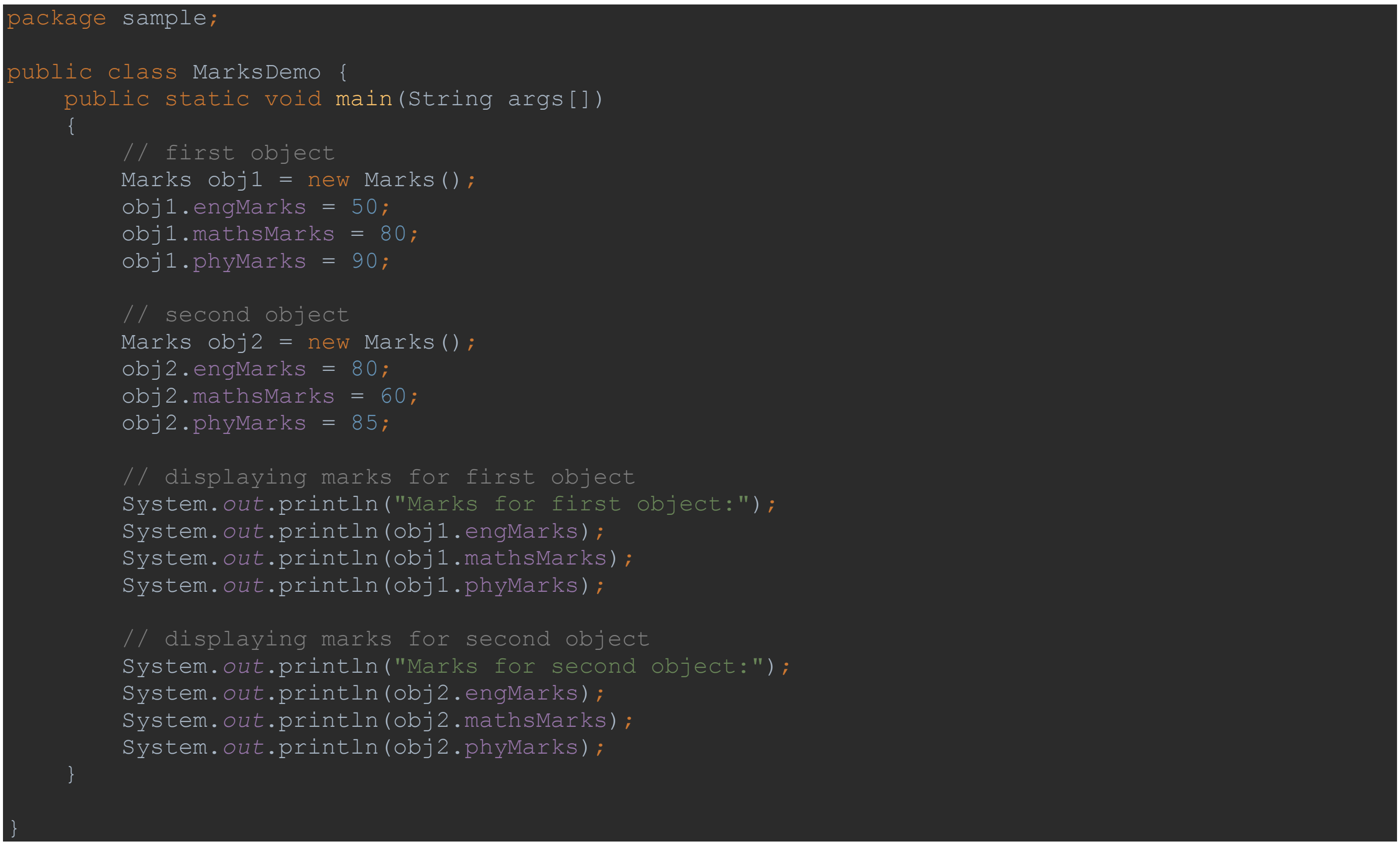
```
▼ D:\comjavafoundation\src\test\java\sample\StudentDetails.java
      Error:(12, 50)  java: cannot find symbol
                        symbol:   variable age
                        location: class sample.StudentDetails
```

INSTANCE VARIABLES: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

Sample Program:

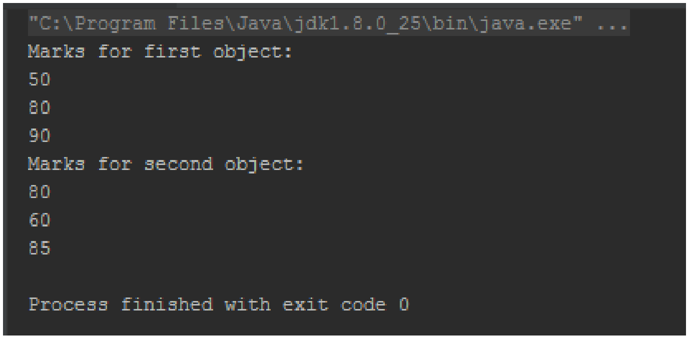


```
package sample;
public class Marks {
    // These variables are instance variables.
    // These variables are in a class
    // and are not inside any function
    int engMarks;
    int mathsMarks;
    int phyMarks;
}
```



```
package sample;

public class MarksDemo {
    public static void main(String args[])
    {
        // first object
        Marks obj1 = new Marks();
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 90;

        // second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;
        obj2.phyMarks = 85;

        // displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);
        System.out.println(obj1.phyMarks);

        // displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
        System.out.println(obj2.phyMarks);
    }

}
```

Output:



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Marks for first object:
50
80
90
Marks for second object:
80
60
85

Process finished with exit code 0
```

STATIC VARIABLES: Static variables are also known as Class variables.

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- If we access the static variable without the class name, Compiler will automatically append the class name.

To access static variables, we need not create an object of that class, we can simply access the variable as

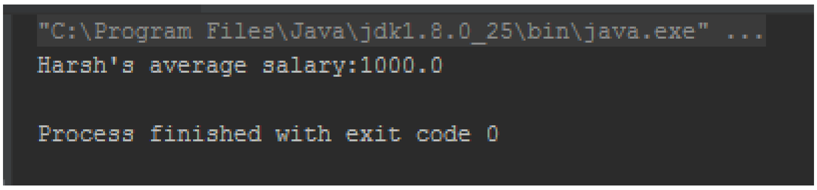
```
class_name.variable_name;
```

Sample Program:

```
package sample;
public class Emp {
    // static variable salary
    public static double salary;
    public static String name = "Harsh";
}
```

```
package sample;
public class EmpDemo {
    public static void main(String args[])
    {
        // accessing static variable without object
        Emp.salary = 1000;
        System.out.println(Emp.name + "'s average salary:"
                + Emp.salary);
    }
}
```

Output:



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Harsh's average salary:1000.0

Process finished with exit code 0
```

SCOPE OF VARIABLES IN JAVA

Scope of a variable is the part of the program where the variable is accessible.

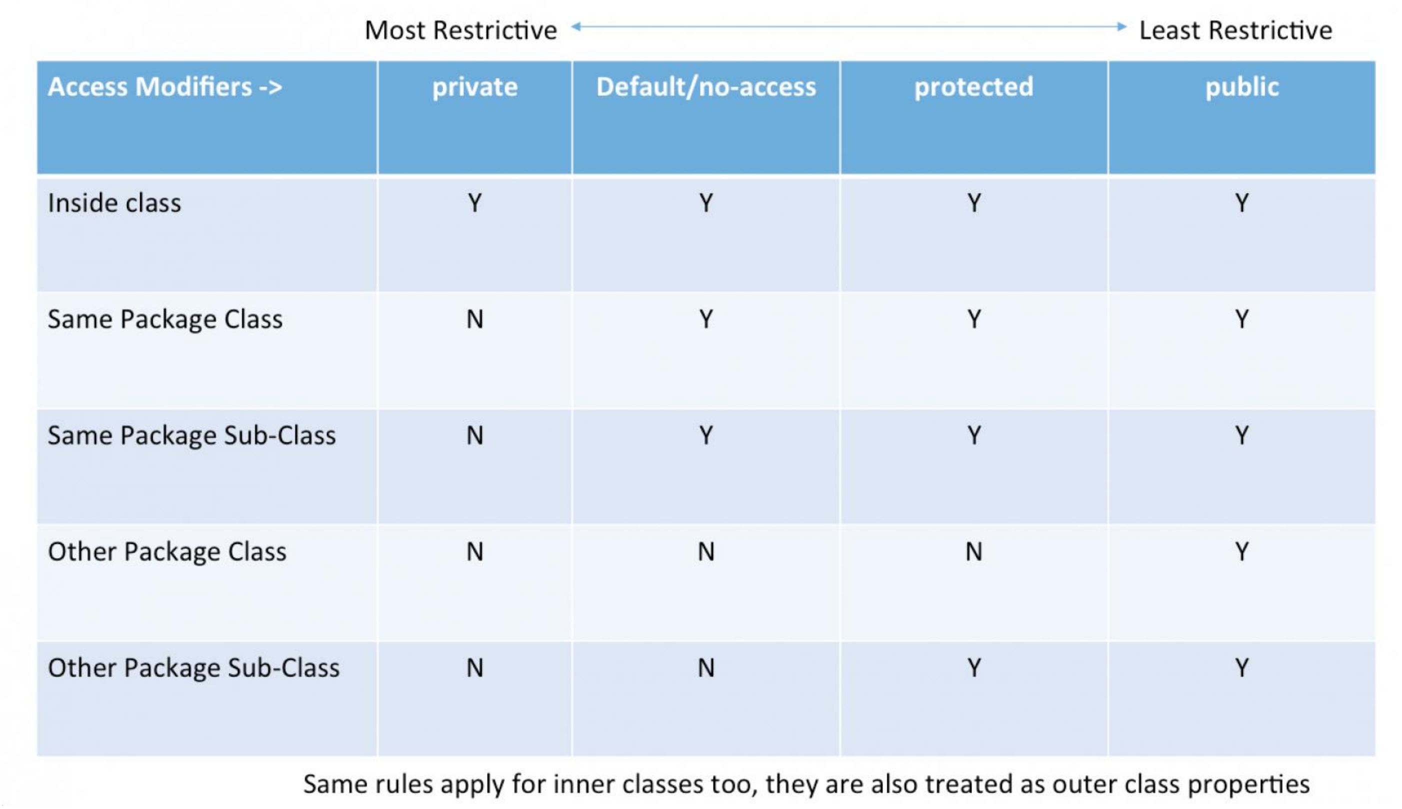
Member Variables (Class Level Scope)

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
package sample;
public class Test {
    // All variables defined directly inside a class
    // are member variables
    int a;
    private String b
    void method1() {....}
    int method2() {....}
    char c;
}
```

We can declare class variables anywhere in class, but outside methods.

- Member variables can be accessed outside a class with following rules



Most Restrictive ← → Least Restrictive

Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Same rules apply for inner classes too, they are also treated as outer class properties

Local Variables (Method Level Scope)

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
package sample;
public class Test {
    void method1()
    {
        // Local variable (Method level scope)
        int x;
    }
}
```

Note: Local variables don't exist after method's execution is over.

Loop Variables (Block Scope)

A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

```
package sample;
public class Test {
    public static void main(String args[])
    {
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }
        // Uncommenting below line would produce
        // error since variable x is out of scope.
        // System.out.println(x);
    }
}
```

BLANK FINAL

A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

```
final int i = 10;
i = 30; // Error because i is final.
```

A **blank final** variable in Java is a final variable that is not initialized during declaration. Below is a simple example of blank final.

```
// A simple blank final example
        final int i;
        i = 30;
```

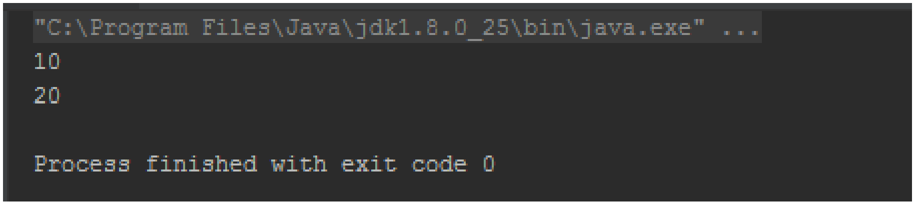
HOW ARE VALUES ASSIGNED TO BLANK FINAL MEMBERS OF OBJECTS?

Values must be assigned in constructor.

```
 package sample;
// A sample Java program to demonstrate use and
// working of blank final

public class Test {

    // We can initialize here, but if we
    // initialize here, then all objects get
    // the same value.  So we use blank final
    final int i;

    Test(int x)
    {
        // Since we have initialized above, we
        // must initialize i in constructor.
        // If we remove this line, we get compiler
        // error.
        i = x;
    }
}
```

```
package sample;
// Driver Code
public class Main {
    public static void main(String args[])
    {
        Test t1 = new Test(10);
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

Output:



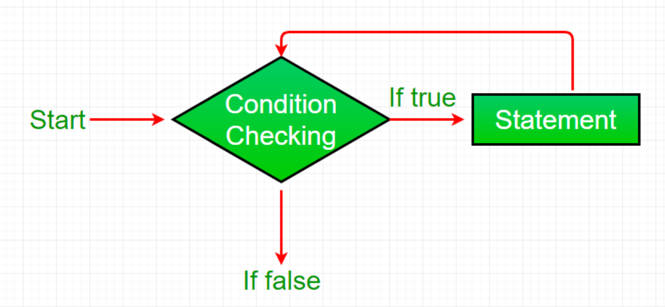
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
10
20

Process finished with exit code 0
```

WHILE LOOP - FOR LOOP - DO WHILE

while loop: A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

Syntax:

```
while (boolean condition)
{
    loop statements...
}
```

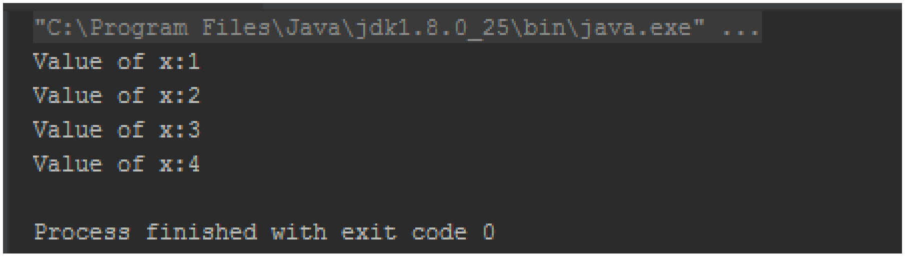


Once the condition is evaluated to true, the statements in the loop body are executed.
When the condition becomes false, the loop terminates which marks the end of its life cycle.

JAVA PROGRAM TO ILLUSTRATE WHILE LOOP

```
package sample;

public class WhileLoopDemo {
    public static void main(String args[])
    {
        int x = 1;

        // Exit when x becomes greater than 4
        while (x <= 4)
        {
            System.out.println("Value of x:" + x);

            // Increment the value of x for
            // next iteration
            x++;
        }
    }
}
```

Output:

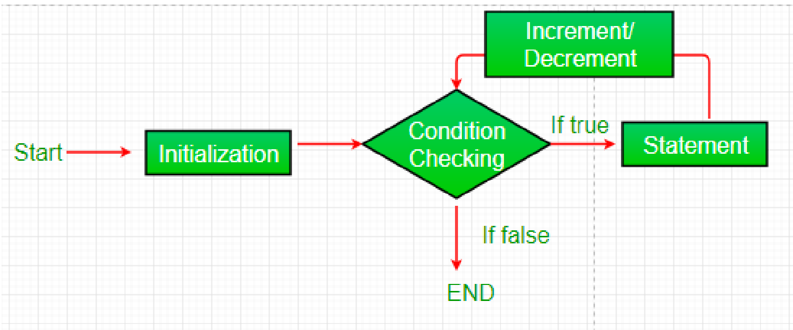


```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Value of x:1
Value of x:2
Value of x:3
Value of x:4

Process finished with exit code 0
```

for loop: a for statement consumes the initialization, condition and increment/decrement in one line.

Syntax:

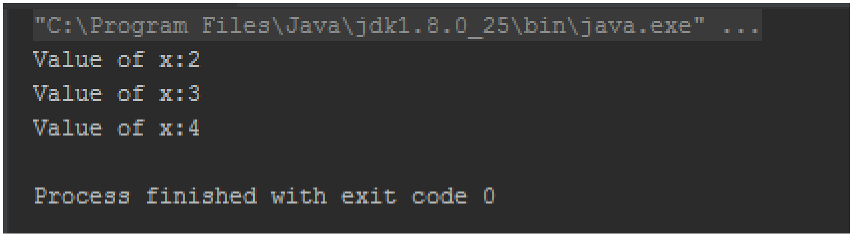
```
for (initialization condition; testing condition;
increment/decrement)
{
    statement(s)
}
```



1. **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
2. **Testing Condition:** It is used for testing the exit condition for a loop. It must return a Boolean value.
3. **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
4. **Increment/ Decrement:** It is used for updating the variable for next iteration.
5. **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle.

```
package sample;
// Java program to illustrate for loop.
public class ForLoopDemo {
    public static void main(String args[])
    {
        // for loop begins when x=2
        // and runs till x <=4
        for (int x = 2; x <= 4; x++)
            System.out.println("Value of x:" + x);
    }
}
```

Output

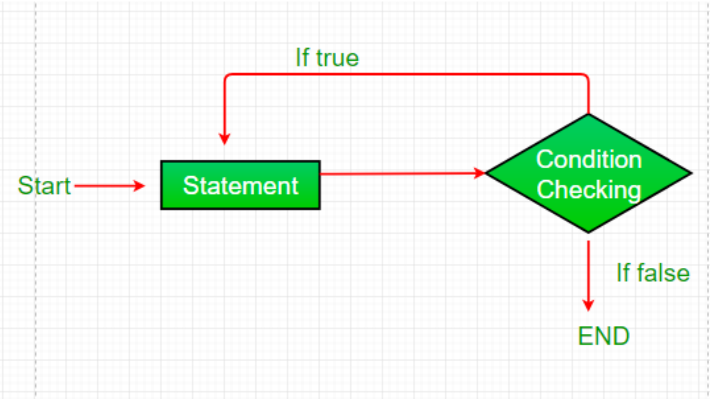


```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Value of x:2
Value of x:3
Value of x:4

Process finished with exit code 0
```

do while: do while loop is similar to while loop with only difference that it checks for condition after executing the statements

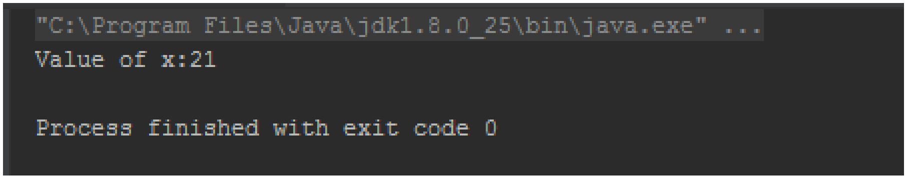
Syntax:

```
do
    {
        statements..
    }
while (condition);
```



```
package sample;
// Java program to illustrate do-while loop
public class DowhileloopDemo {
    public static void main(String args[])
    {
        int x = 21;
        do
        {
            // The line will be printed even
            // if the condition is false
            System.out.println("Value of x:" + x);
            x++;
        }
        while (x < 20);
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Value of x:21

Process finished with exit code 0
```

DECISION MAKING - IF | IF-ELS+E | NESTED-IF | IF-ELSE-IF | SWITCH-CASE |JUMP

Decision making in programming is similar to decision making in real life. In programming also we face some situations where we want a certain block of code to be executed when some condition is fulfilled.

if: if statement is the most simple decision making statement. It is used if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
        {
        // Statements to execute if
        // condition is true
        }
```

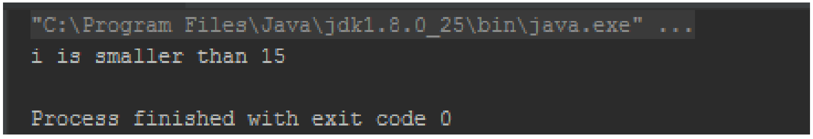
if-else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
        {
        // Executes this block if
        // condition is true
        }
        else
        {
        // Executes this block if
        // condition is false
        }
```

```
package sample;
// Java program to illustrate if-else statement
public class IfElseDemo {
    public static void main(String args[])
    {
        int i = 10;

        if (i < 15)
            System.out.println("i is smaller than 15");
        else
            System.out.println("i is greater than 15");
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
i is smaller than 15

Process finished with exit code 0
```

nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements means an if statement inside an if statement.

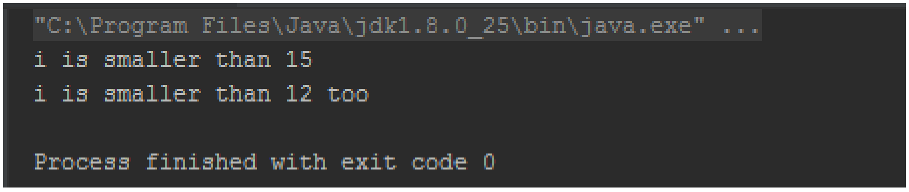
Nested if statements means an if statement inside an if statement.
i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
         {
         // Executes when condition1 is true
         if (condition2)
         {
         // Executes when condition2 is true
         }
         }
```

```
package sample;
// Java program to illustrate nested-if statement
public class NestedIfDemo {
    public static void main(String args[])
    {
        int i = 10;

        if (i == 10)
        {
            // First if statement
            if (i < 15)
                System.out.println("i is smaller than 15");

            // Nested - if statement
            // Will only be executed if statement above
            // it is true
            if (i < 12)
                System.out.println("i is smaller than 12 too");
            else
                System.out.println("i is greater than 15");
        }
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
i is smaller than 15
i is smaller than 12 too

Process finished with exit code 0
```

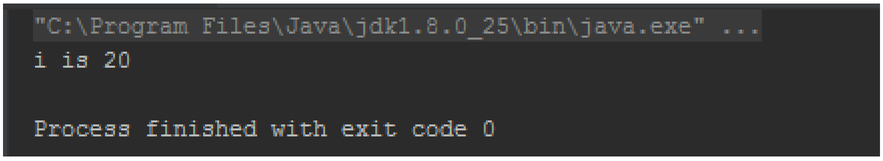
if-else-if ladder: The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:

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

```
package sample;
// Java program to illustrate if-else-if ladder
public class IfelseifDemo {
    public static void main(String args[])
    {
        int i = 20;

        if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
            System.out.println("i is 15");
        else if (i == 20)
            System.out.println("i is 20");
        else
            System.out.println("i is not present");
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
i is 20

Process finished with exit code 0
```

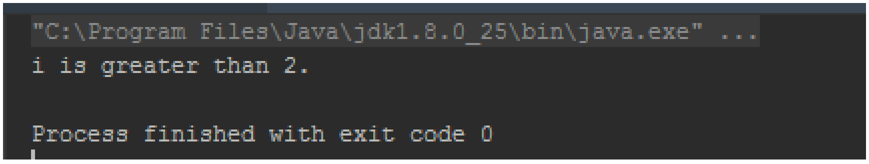
switch-case The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax:

```
switch (expression)
        {
        case value1:
        statement1;
        break;
        case value2:
        statement2;
        break;
        .
        .
        case valueN:
        statementN;
        break;
default:
        statementDefault;
        }
```

```
package sample;
// Java program to illustrate switch-case
public class SwitchCaseDemo {
    public static void main(String args[])
    {
        int i = 9;
        switch (i)
        {
            case 0:
                System.out.println("i is zero.");
                break;
            case 1:
                System.out.println("i is one.");
                break;
            case 2:
                System.out.println("i is two.");
                break;
            default:
                System.out.println("i is greater than 2.");
        }
    }
}
```

Output:



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
i is greater than 2.

Process finished with exit code 0
```

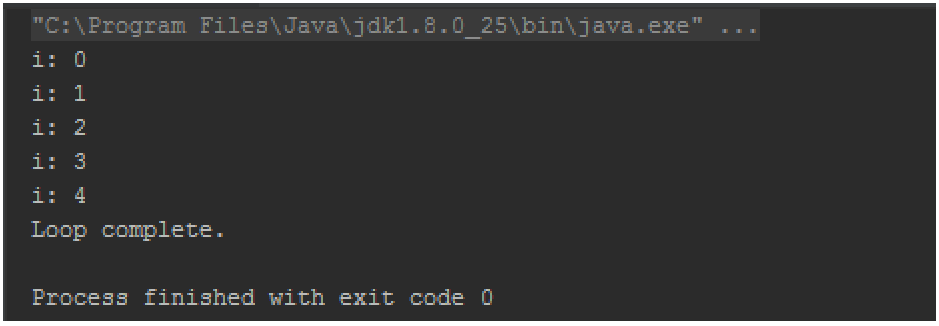
jump: Java supports three jump statement: **break, continue** and **return**. These three statements transfer control to other part of the program.

BREAK: In Java, break is majorly used for:

- Terminate a sequence in a switch statement (discussed above).
- To exit a loop.

```
package sample;
// Java program to illustrate using
// break to exit a loop
public class BreakLoopDemo {
    public static void main(String args[])
    {
        // Initially loop is set to run from 0-9
        for (int i = 0; i < 10; i++)
        {
            // terminate loop when i is 5.
            if (i == 5)
                break;

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

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
i: 0
i: 1
i: 2
i: 3
i: 4
Loop complete.

Process finished with exit code 0
```

Using break as a Form of Goto

Java uses label. A Label is use to identifies a block of code.

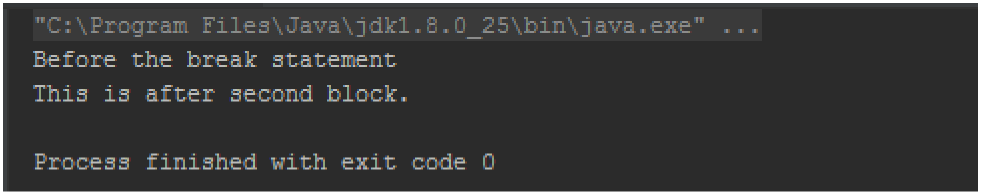
Syntax:

```
label:
        {
        statement1;
        statement2;
        statement3;
        .
        .
        }
```

Now, break statement can be used to jump out of target block.

```
package sample;
// Java program to illustrate using break with goto
public class BreakLabelDemo {
    public static void main(String args[])
    {
        boolean t = true;

        // label first
        first:
        {
            // Illegal statement here as label second is not
            // introduced yet break second;
            second:
            {
                third:
                {
                    // Before break
                    System.out.println("Before the break statement");

                    // break will take the control out of
                    // second label
                    if (t)
                        break second;
                    System.out.println("This won't execute.");
                }
                System.out.println("This won't execute.");
            }

            // Third block
            System.out.println("This is after second block.");
        }
    }
}
```

Output



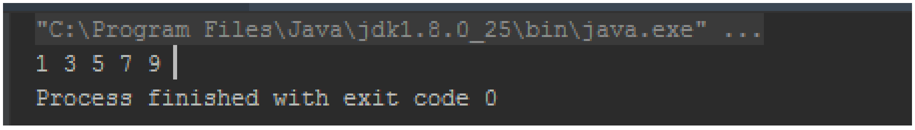
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Before the break statement
This is after second block.

Process finished with exit code 0
```

Continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.

```
package sample;
// Java program to illustrate using
// continue in an if statement

public class ContinueDemo {
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            // If the number is even
            // skip and continue
            if (i%2 == 0)
                continue;

            // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```

Output

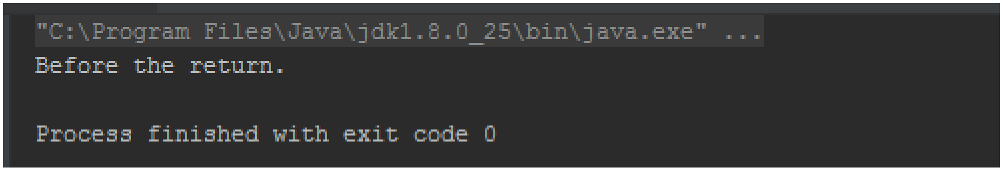


```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
1 3 5 7 9 
Process finished with exit code 0
```

Return: The return statement is used to explicitly return from a method.

```
package sample;
// Java program to illustrate using return
public class Return {
    public static void main(String args[])
    {
        boolean t = true;
        System.out.println("Before the return.");

        if (t)
            return;

        // Compiler will bypass every statement
        // after return
        System.out.println("This won't execute.");
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Before the return.

Process finished with exit code 0
```

COMMENTS

Proper use of comments makes maintenance easier and finding bugs easily. Comments are ignored by the compiler while compiling a code.
In Java there are three types of comments:

1. *Single – line comments.*
2. *Multi – line comments.*
3. *Documentation comments.*

Single-line Comments

Syntax:

```
//Comments here( Text in this line only is considered as comment )
```

```
package sample;
//Java program to show single line comments
public class Comment {
    public static void main(String args[])
    {
        // Single line comment here
        System.out.println("Single line comment above");
    }
}
```

Multi-line Comments

Syntax:

```
/*Comment starts
continues
continues
.
.
.
Comment ends*/
```

```
package sample;
//Java program to show multi line comments
public class MultiLineComments {
    public static void main(String args[])
    {
        System.out.println("Multi line comments below");
      /*Comment line 1
      Comment line 2
      Comment line 3*/
    }
}
```

Documentation Comments

It helps to generate a documentation page for reference

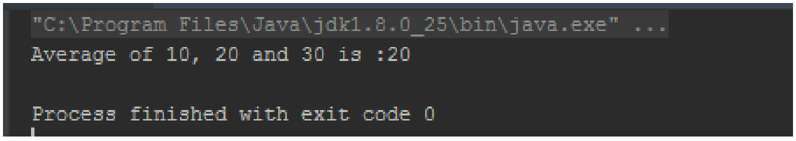
Syntax:

```
/**Comment start
 *
 *tags are used in order to specify a parameter
 *or method or heading
 *HTML tags can also be used
 *such as <h1>
 *
 *comment ends*/
```

Available tags to use: @author {@code} {@docRoot} @deprecated @exception {@link} @param @return @throws {@value} @version

```
package sample;
//Java program to illustrate frequently used
// Comment tags

/**
 * <h1>Find average of three numbers!</h1>
 * The FindAvg program implements an application that
 * simply calculates average of three integers and Prints
 * the output on the screen.
 *
 * @author Pratik Agarwal
 * @version 1.0
 * @since 2017-02-18
 */

public class FindAvg {
    /**
     * This method is used to find average of three integers.
     * @param numA This is the first parameter to findAvg method
     * @param numB This is the second parameter to findAvg method
     * @param numC This is the second parameter to findAvg method
     * @return int This returns average of numA, numB and numC.
     */
    public int findAvg(int numA, int numB, int numC)
    {
        return (numA + numB + numC)/3;
    }

    /**
     * This is the main method which makes use of findAvg method.
     * @param args Unused.
     * @return Nothing.
     */

    public static void main(String args[])
    {
        FindAvg obj = new FindAvg();
        int avg = obj.findAvg(10, 20, 30);

        System.out.println("Average of 10, 20 and 30 is :" + avg);
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Average of 10, 20 and 30 is :20

Process finished with exit code 0
```

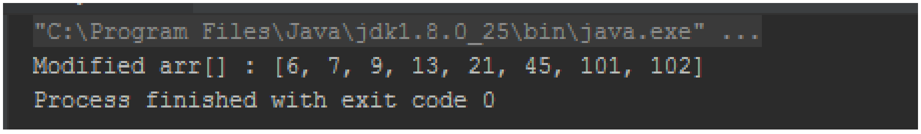
ARRAYS AND COLLECTIONS SORT

There are two in-built methods to sort in Java.

Arrays.Sort() works for arrays which can be of primitive data type.

```java
package sample;
// A sample Java program to demonstrate working of
// Arrays.sort().
// It by default sorts in ascending order.
import java.util.Arrays;
public class ArraysSort {
    public static void main(String[] args)
    {
        int[] arr = { 13, 7, 6, 45, 21, 9, 101, 102 };
        Arrays.sort(arr);
        System.out.printf("Modified arr[] : %s",
                Arrays.toString(arr));
    }
}
```

Output

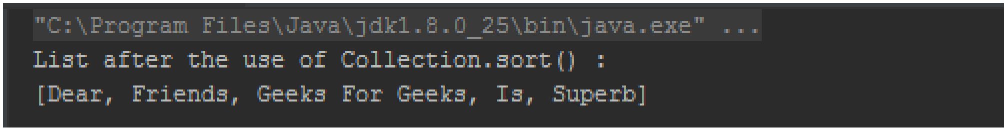


```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]
Process finished with exit code 0
```

Collections.sort() works for objects Collections like ArrayList and LinkedList.

```java
package sample;
// Java program to demonstrate working of Collections.sort()
import java.util.*;

public class CollectionsSort {
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

      /* Collections.sort method is sorting the
      elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of"
                + " Collection.sort() :\n" + al);
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
List after the use of Collection.sort() :
[Dear, Friends, Geeks For Geeks, Is, Superb]
```

CLASSES AND OBJECTS

Class

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. MODIFIERS : A class can be public or has default access
2. CLASS NAME: The name should begin with an initial letter (capitalized by convention).

3. SUPERCLASS (IF ANY): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. INTERFACES (IF ANY): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. BODY: The class body surrounded by braces, { }.

Object

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects.

1. STATE: It is represented by attributes of an object. It also reflects the properties of an object.
2. BEHAVIOR: It is represented by methods of an object. It also reflects the response of an object with other objects.
3. IDENTITY: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog

Identity	**State/Attributes**	**Behaviors**
Name of dog	*Breed* *Age* *Color*	*Bark* *Sleep* *Eat*

```java
package sample;
// Class Declaration
public class Dog {
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
               int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }

    // method 3
    public int getAge()
    {
        return age;
    }

    // method 4
    public String getColor()
    {
        return color;
    }
    @Override
    public String toString()
    {
        return("Hi my name is "+ this.getName()+
                ".\nMy breed,age and color are " +
                this.getBreed()+"," + this.getAge()+
                ","+ this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }

}
```

WAYS TO CREATE OBJECT OF A CLASS

Using new keyword: It is the most common and general way to create object in java.

Example:

```
// creating object of class Test
Test t = new Test();
```

Using Class.forName(String className) method : There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name.

```
// creating object of public class Test
// consider class Test present in com.p1 package
    Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

Using clone() method: clone() method is present in Object class. It creates and returns a copy of the object.

```
// creating object of class Test
Test t1 = new Test();
// creating clone of above object
Test t2 = (Test)t1.clone();
```

Deserialization: De-serialization is technique of reading an object from the saved state in a file.

```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

INHERTANCE

Inheritance is an important pillar of OOP (Object Oriented Programming)

Important terminology:

- **Super Class:** The class whose features are inherited is known as super class (or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as sub class (or a derived class, extended class, or child class).

How to use inheritance in Java

The keyword used for inheritance is ***extends****.*

Syntax:

```
class derived-class extends base-class
{
        //methods and fields
        }
```

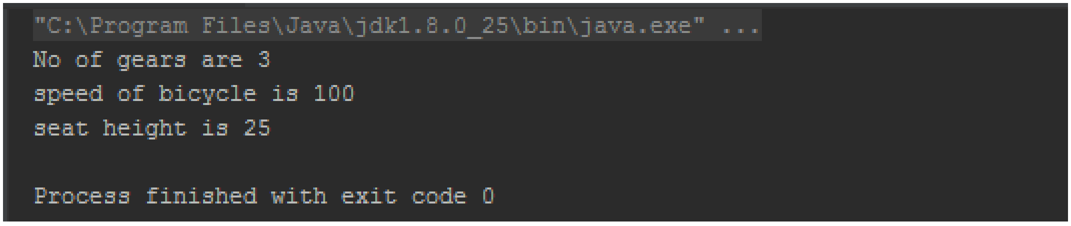
Example: In below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class which extends Bicycle class and class Test is a driver class to run program.

```
package sample;
//Java program to illustrate the
// concept of inheritance
// base class
public class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return("No of gears are "+gear
                +"\n"
                + "speed of bicycle is "+speed);
    }
}
```

```
package sample;
// derived class
public class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }
}
```

```
package sample;
// driver class
public class TestDriverClass {
    public static void main(String args[])
    {
        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

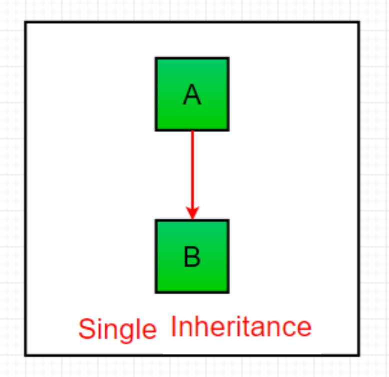
Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
No of gears are 3
speed of bicycle is 100
seat height is 25

Process finished with exit code 0
```

Types of Inheritance in Java

SINGLE INHERITANCE: In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

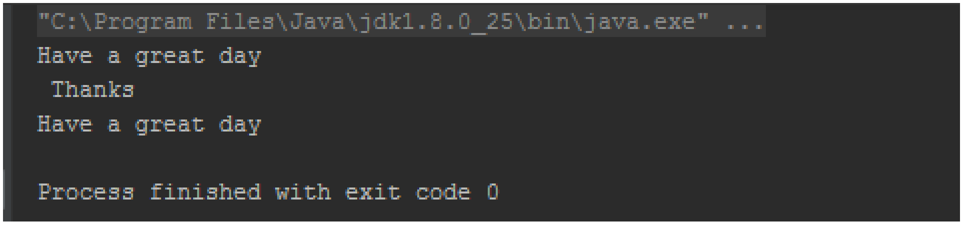


```
package inheritanceInJava;
//Java program to illustrate the
// concept of single inheritance
import java.lang.*;
public class one {
    public void print()
    {
        System.out.println("Have a great day");
    }
}
```

```
package inheritanceInJava;

public class two extends one
{
    public void print_for()
    {
        System.out.println(" Thanks");
    }
}
```

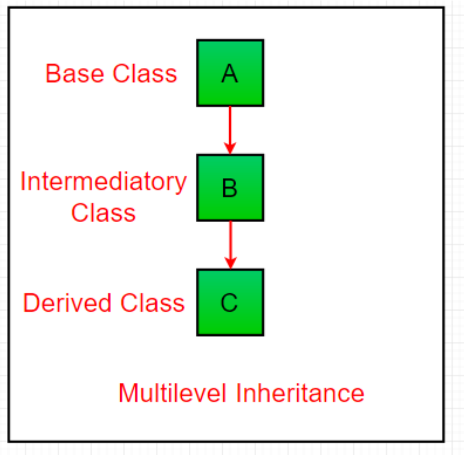
```
package inheritanceInJava;
// Driver class
public class Main {
    public static void main(String[] args)
    {
        two g = new two();
        g.print();
        g.print_for();
        g.print();
    }
}
```

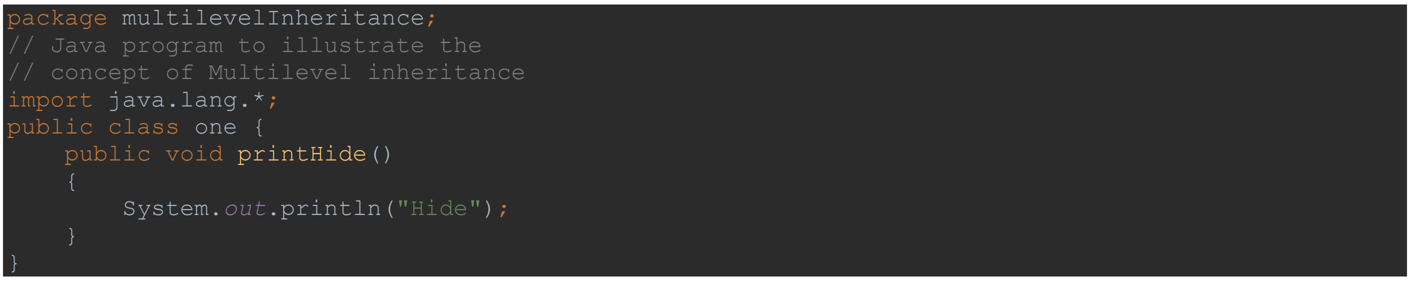
Output



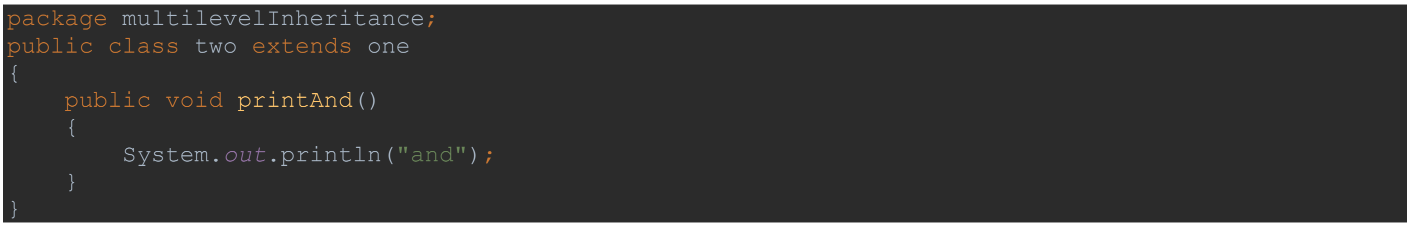
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Have a great day
 Thanks
Have a great day

Process finished with exit code 0
```

MULTILEVEL INHERITANCE: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

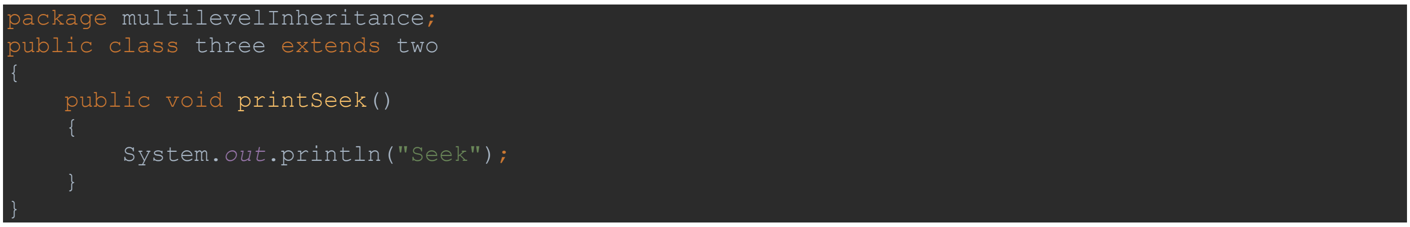




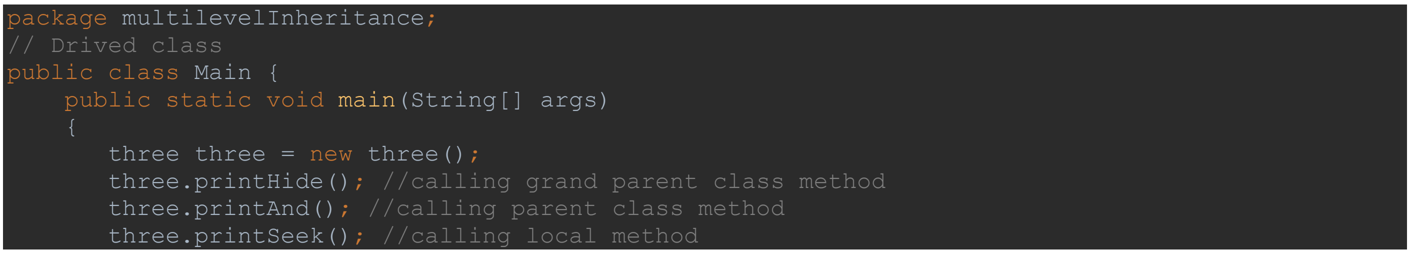
```
package multilevelInheritance;
// Java program to illustrate the
// concept of Multilevel inheritance
import java.lang.*;
public class one {
    public void printHide()
    {
        System.out.println("Hide");
    }
}
```



```
package multilevelInheritance;
public class two extends one
{
    public void printAnd()
    {
        System.out.println("and");
    }
}
```

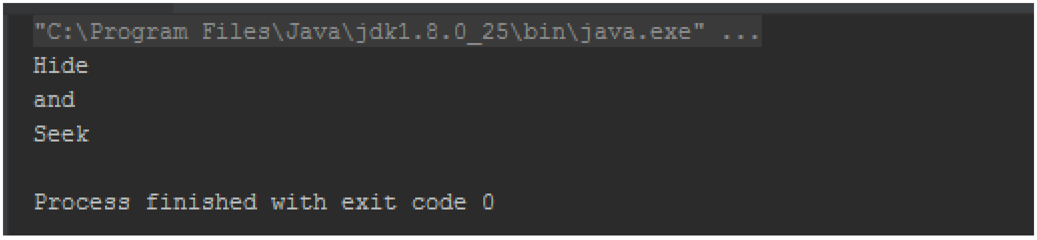


```
package multilevelInheritance;
public class three extends two
{
    public void printSeek()
    {
        System.out.println("Seek");
    }
}
```



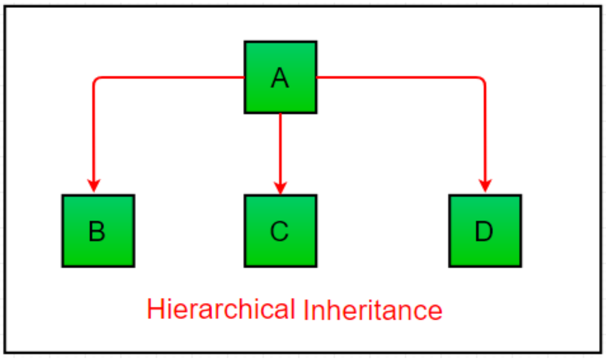
```
package multilevelInheritance;
// Drived class
public class Main {
    public static void main(String[] args)
    {
       three three = new three();
       three.printHide(); //calling grand parent class method
       three.printAnd(); //calling parent class method
       three.printSeek(); //calling local method
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Hide
and
Seek

Process finished with exit code 0
```

HIERARCHICAL INHERITANCE: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class. In below image, the class A serves as a base class for the derived class B, C and D.



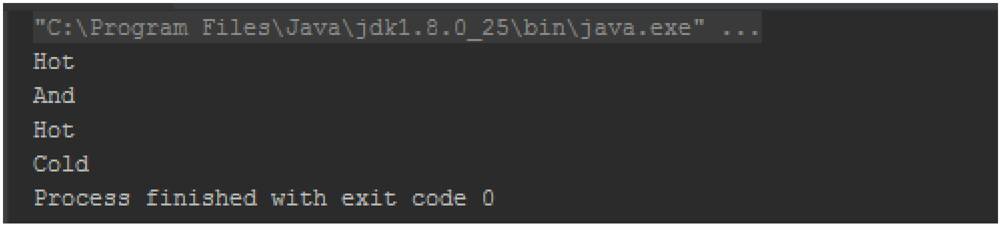
```
package hierarchicalInheritance;
// Java program to illustrate the
// concept of Hierarchical inheritance

public class One {
    public void printHot()
    {
        System.out.println("Hot");
    }
}
```

```
package hierarchicalInheritance;

public class Two extends One {
    public void printAnd()
    {
        System.out.println("And");
    }
}
```

```
package hierarchicalInheritance;

public class Three extends One {
   public void printCold (){
       System.out.printf("Cold");
   }
}
```

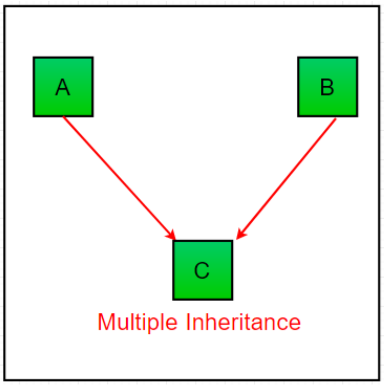
```
package hierarchicalInheritance;
// Drived class
public class Main {
    public static void main (String arg[]){
        Two two = new Two();
        Three three = new Three();
        two.printHot(); //calling parent class method
        two.printAnd(); //calling local class method
        three.printHot(); //calling parent class method
        three.printCold(); //calling local class method
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Hot
And
Hot
Cold
Process finished with exit code 0
```

MULTIPLE INHERITANCE (THROUGH INTERFACES): In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



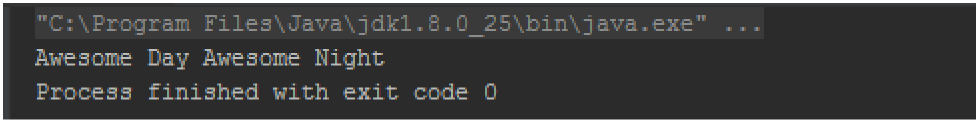
```
package multipleIinheritance;

public interface One {
    public void printDay();
}
```

```
package multipleIinheritance;

public interface Two {
    public void printNight();
}
```

```
package multipleIinheritance;

public interface Three extends One, Two {
    public void printAwesome ();
}
```

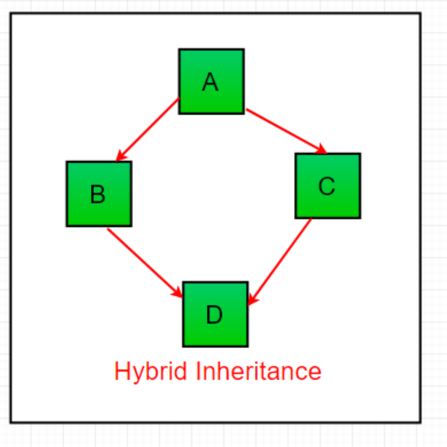
```
package multipleIinheritance;
// Java program to illustrate the
// concept of Multiple inheritance

public class Main implements Three {
    public void printAwesome() {
        System.out.printf("Awesome ");
    }

    public void printDay() {
        System.out.printf("Day ");
    }

    public void printNight() {
        System.out.printf("Night ");
    }
    public static void main (String arg[]){
        Main main = new Main();
        main.printAwesome(); // Calling Parent Interface Method
        main.printDay(); // Calling local Interface Method
        main.printAwesome(); // Calling Parent Interface Method
        main.printNight(); // Calling local Interface Method
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Awesome Day Awesome Night
Process finished with exit code 0
```

HYBRID INHERITANCE (THROUGH INTERFACES): It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



ENCAPSULATION

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as `private` (only accessible within the same class)
- provide public **setter** and **getter** methods to access and update the value of a `private` variable

Get and Set

`private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **getter** and **setter** methods.

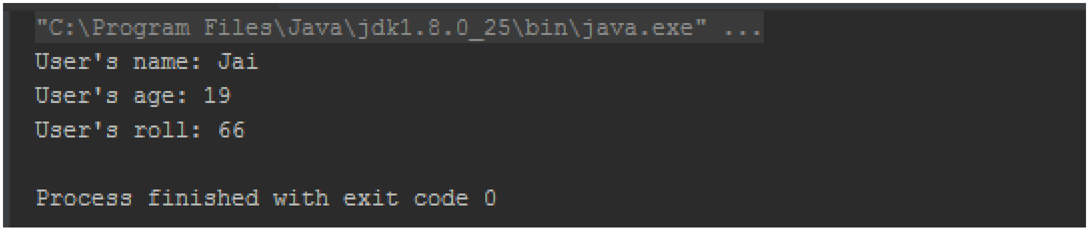
The `get` method returns the variable value, and the `set` method sets the value.

```
package encapsulate;
// Java program to demonstrate encapsulation
public class Encapsulate {
    // private variables declared
    // these can only be accessed by
    // public methods of class
    private String userName;
    private int userRoll;
    private int userAge;

    // get method for age to access
    // private variable userAge
    public int getAge()
    {
        return userAge;
    }

    // get method for name to access
    // private variable userName
    public String getName()
    {
        return userName;
    }

    // get method for roll to access
    // private variable userRoll
    public int getRoll()
    {
        return userRoll;
    }
    // set method for age to access
    // private variable userage
    public void setAge( int newAge)
    {
        userAge = newAge;
    }

    // set method for name to access
    // private variable userName
    public void setName(String newName)
    {
        userName = newName;
    }

    // set method for roll to access
    // private variable userRoll
    public void setRoll( int newRoll)
    {
        userRoll = newRoll;
    }

}
```

The program to access variables of the class EncapsulateDemo is shown below:

```
package encapsulate;

public class TestEncapsulation {
    public static void main (String[] args)
    {
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Jai");
        obj.setAge(19);
        obj.setRoll(66);

        // Displaying values of the variables
        System.out.println("User's name: " + obj.getName());
        System.out.println("User's age: " + obj.getAge());
        System.out.println("User's roll: " + obj.getRoll());

        // Direct access of userRoll is not possible
        // due to encapsulation
        // System.out.println("User's roll: " + obj.userName);
    }

}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
User's name: Jai
User's age: 19
User's roll: 66

Process finished with exit code 0
```

ABSTRACTION

Data **abstraction** is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
package abstraction;

public abstract class Animal {
    public abstract void animalSound();
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class. Let's convert the Animal class

Abstract class

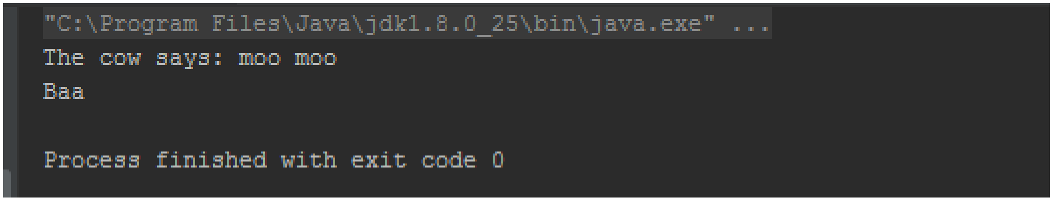
```
package abstraction;

public abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sheep() {
        System.out.println("Baa");
    }

}
```

Subclass (inherit from Animal)

```
package abstraction;

public class Cow extends Animal{

    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("he cow says: moo moo");
    }
}
```

```
package abstraction;

public class MyMainClass {
    public static void main(String[] args) {
        Cow cow = new Cow(); // Create a Cow object
        cow.animalSound();
        cow.sheep();
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
The cow says: moo moo
Baa

Process finished with exit code 0
```

POLYMORPHISM

Polymorphism means "many forms", and it occurs when we have many classes that related to each other by inheritance. **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called `Animal` that has a method called `animalSound ()`

Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the dog Woof, and the cat meows, etc.):

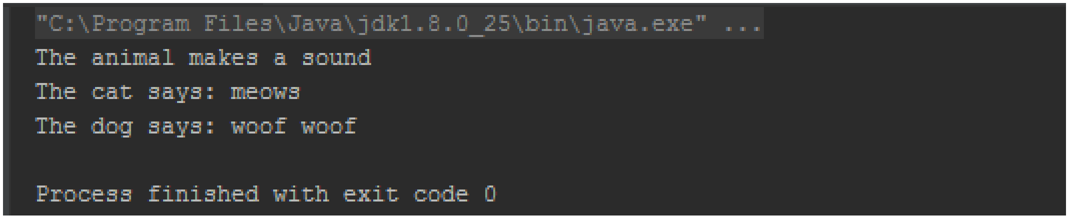
```
package polymorphism;

public class Animal {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}
```

```
package polymorphism;

public class Dog extends Animal {
    public void animalSound() {
        System.out.println("The dog says: woof woof");
    }
}
```

```
package polymorphism;

public class Cat extends Animal {
    public void animalSound() {
        System.out.println("The cat says: meows");
    }
}
```

```
package polymorphism;

public class MyMainClass {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();  // Create a Animal object
        Animal myCat = new Cat();  // Create a Cat object
        Animal myDog = new Dog();  // Create a Dog object

        myAnimal.animalSound();
        myCat.animalSound();
        myDog.animalSound();
    }
}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
The animal makes a sound
The cat says: meows
The dog says: woof woof

Process finished with exit code 0
```

METHOD OVERLOADING AND METHOD OVERRIDING

No.	Method Overloading	Method Overriding
1)	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2)	Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3)	In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
4)	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.

Method Overloading example

```
package overloading;

public class OverloadingExample {
    static int add(int a,int b)
    {
        return a+b;
    }
    static int add(int a,int b,int c)
    {
        return a+b+c;
    }
}
```

Method Overriding example

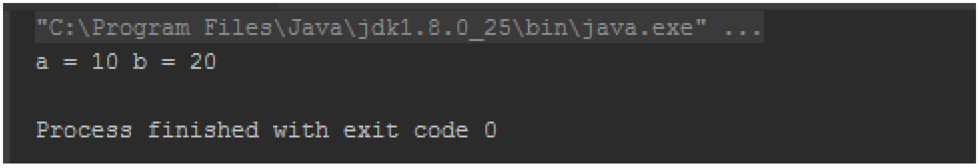
```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
}
```

THIS KEYWORD

'this' is a reference variable that refers to the current object.

```
package thisReference;
//Java code for using 'this' keyword to
//refer current class instance variables

public class TestThisReference {
    int a;
    int b;
    // Parameterized constructor
    TestThisReference(int a, int b)
    {
        this.a = a;
        this.b = b;
    }
    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + " b = " + b);
    }
    public static void main(String[] args)
    {
        TestThisReference object = new TestThisReference(10, 20);
        object.display();
    }

}
```

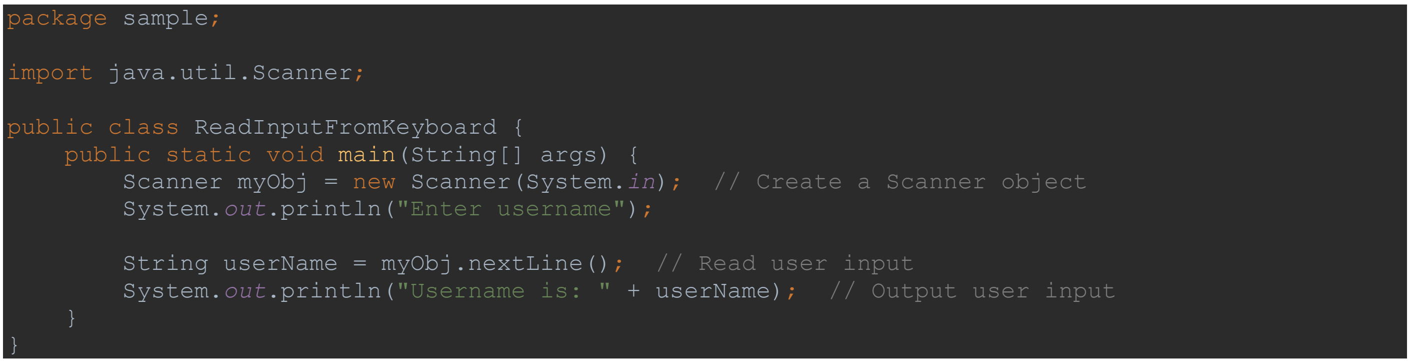
Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
a = 10 b = 20

Process finished with exit code 0
```

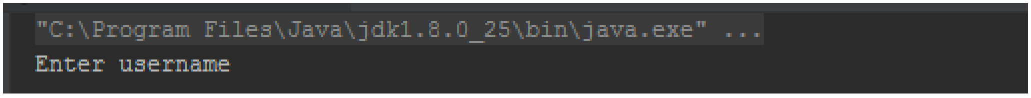
USER INPUT

The `Scanner` class is used to get user input, and it is found in the `java.util` package.
In our example, we will use the `nextLine()` method, which is used to read Strings:



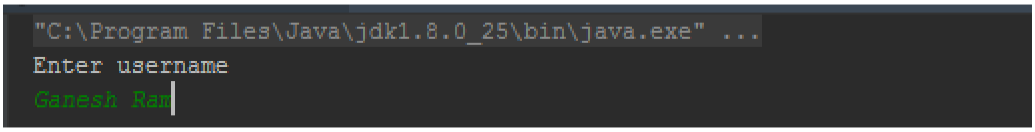
```java
package sample;

import java.util.Scanner;

public class ReadInputFromKeyboard {
    public static void main(String[] args) {
        Scanner myObj = new Scanner(System.in);  // Create a Scanner object
        System.out.println("Enter username");

        String userName = myObj.nextLine();  // Read user input
        System.out.println("Username is: " + userName);  // Output user input
    }
}
```

Input prompt will display



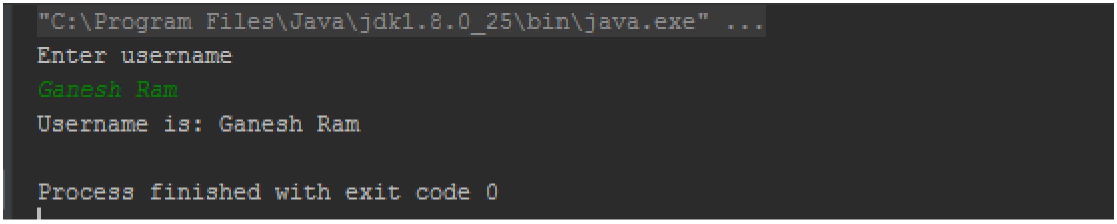
```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Enter username
```

Enter User Name Now



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Enter username
Ganesh Ram
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
Enter username
Ganesh Ram
Username is: Ganesh Ram

Process finished with exit code 0
```

INPUT TYPES

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

ARRAYLIST

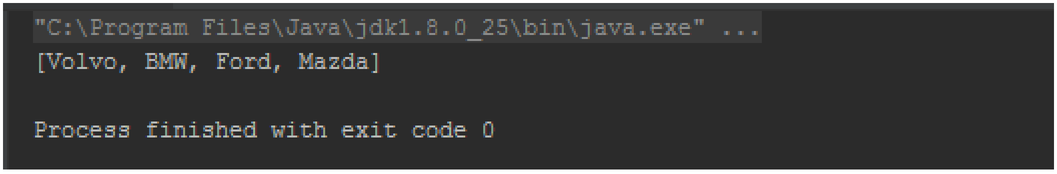
The `ArrayList` class is a resizable `array`, which can be found in the `java.util` package.

Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

```
package sample;

import java.util.ArrayList;

public class MyClass {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);
    }

}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
[Volvo, BMW, Ford, Mazda]

Process finished with exit code 0
```

Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

Example

```
cars.get(0);
```

Change an Item

To modify an element, use the `set()` method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

Example

```
cars.remove(0);
```

To remove all the elements in the `ArrayList`, use the `clear()` method:

Example

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList have, use the `size` method:

Example

```
cars.size();
```

HASHMAP

You learned from the previous chapter, that Arrays store items as an ordered collection, and you have to access them with an index number (`int` type). A `HashMap` however, store items in "**key**/**value**" pairs, and you can access them by an index of another type (e.g. a `String`).

One object is used as a key (index) to another object (value). It can store different types: `String` keys and `Integer` values, or the same type, like: `String` keys and `String` values:

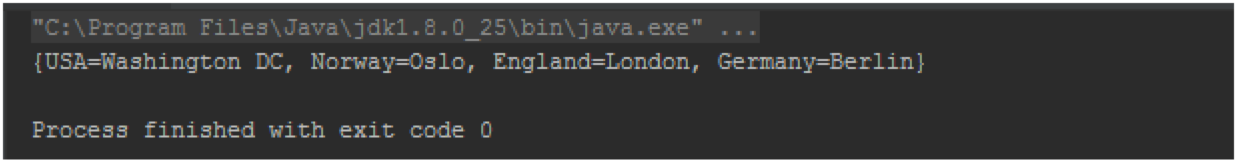
Add Items

The `HashMap` class has many useful methods. For example, to add items to it, use the `put()` method:

Example

```
package sample;
// Import the HashMap class
import java.util.HashMap;
public class MyClass {
    public static void main(String[] args) {

        // Create a HashMap object called capitalCities
        HashMap<String, String> capitalCities = new HashMap<String, String>();

        // Add keys and values (Country, City)
        capitalCities.put("England", "London");
        capitalCities.put("Germany", "Berlin");
        capitalCities.put("Norway", "Oslo");
        capitalCities.put("USA", "Washington DC");
        System.out.println(capitalCities);
    }

}
```

Output



```
"C:\Program Files\Java\jdk1.8.0_25\bin\java.exe" ...
{USA=Washington DC, Norway=Oslo, England=London, Germany=Berlin}

Process finished with exit code 0
```

Access an Item

To access a value in the `HashMap`, use the `get()` method and refer to its key:

Example

```
capitalCities.get("England");
```

Remove an Item

To remove an item, use the `remove()` method and refer to the key:

Example

```
capitalCities.remove("England");
```

To remove all items, use the `clear()` method:

Example

```
capitalCities.clear();
```

HashMap Size

To find out how many items there are, use the `size` method:

Example

```
capitalCities.size();
```